Porting MontaVista Linux to the XUP Virtex-II Pro Development Board

Jonathon W. Donaldson

Advisor: Prof. Warren R. Carithers

Completed: Summer 2005 (054)

August 22, 2006

This Independent Study is submitted in partial fulfillment of the requirements for the degree of

M.S. in Computer Science

Department of Computer Science
Rochester Institute of Technology
102 Lomb Memorial Drive
Rochester, NY 14623
U.S.A.

ABSTRACT

Open source software and hardware create the perfect union when designing for an embedded system. Open source allows a developer to mold the software to conform to the restrictions and capabilities of the hardware which will be executing it. Software that has been crafted for a specific hardware design will also run much smoother and faster as they do not have to consider various unknown conditions. This is a common issue with which general purpose operating systems must contend.

Knowing the advantages of supporting open source software on their embedded systems, many manufacturers have either developed their own open source software in house or partnered with external open source vendors to support their development boards. The latter is the case with Xilinx® Inc. and MontaVista® Software Inc.. Xilinx has joined with MontaVista [1] and other RTOS distributors to specifically support Linux® kernels running on IBM® PowerPC™ cores immersed in the Xilinx V2P and V4FX family of FPGAs.

The first few sections of this document will cover my motivation and objectives for this independent research project as it is closely tied with my master's thesis. The sixth section will cover my approach to this project which doubles as a detailed tutorial. The following two sections discuss some of the difficulties encountered and the lessons learned as a result of completing this project. Finally, I conclude with a few prospects for future development with the board and with what I believe to be the future of open source community as it relates to embedded systems design.

TABLE OF CONTENTS

ABSTRACT **2**

LIST OF ILLUSTRATIONS **5**

1 DISCLAIMER **6**

2 ACKNOWLEDGMENTS **6**

3 INTRODUCTION **6**

4 OBJECTIVES **6**

4.1 PRIMARY OBJECTIVES 7
4.2 SECONDARY OBJECTIVES 7

5 MOTIVATION **8**

6 APPROACH (TUTORIAL) **8**

6.1 RIGHT TOOLS FOR THE RIGHT JOB 9
6.1.1 Hardware *10*
6.1.2 Software *11*
6.2 TESTING THE BOARD 12
6.3 CREATING A SOLID FOUNDATION 12
6.3.1 Configuring the PowerPC Hard Core Processor *12*
6.3.2 Configuring the Peripherals *16*
6.3.3 Configuring the Software Platform *21*
6.4 CREATING THE CROSS-COMPILER ENVIRONMENT 25
6.5 BUILDING THE RTOS KERNEL 26
6.6 TESTING THE KERNEL (W/O RFS) 28
6.7 BUILDING THE RFS 30
6.7.1 Setting up BusyBox *30*
6.7.2 Reconfiguring the Kernel *31*
6.8 TESTING THE COMPLETE SYSTEM 34
6.9 ADDING NETWORKING SUPPORT 34
6.10 SETTING UP AN HTTP SERVER 36
6.11 COMPILING LOCALLY ON THE XUPV2P (FUTURE WORK) 37
6.12 SETTING UP AN SSH SERVER (FUTURE WORK) 37
6.13 BOOTING LINUX VIA NFS (FUTURE WORK) 38
6.14 BOOTING LINUX KERNEL WITH PPCBOOT (FUTURE WORK) 38
6.15 ADDING SUPPORT FOR IPV6 (FUTURE WORK) 38
6.16 RUNNING A SIMPLE IPV6 CHAT CLIENT (FUTURE WORK) 38

7 TRIALS AND TRIBULATIONS **38**

7.1 "`UNDEFINED REFERENCE`" 38
7.2 INFINITE LOOP 38
7.3 DUAL-CORE PROCESSING AND SHARED MEMORY 39
7.4 "`CANNOT OPEN ROOT DEVICE`..." 40
7.5 SPLIT MEMORY VS. LINUX 41
7.6 THE DOS/UNIX EOL PLAGUE 41

8 LESSONS LEARNED **44**

8.1 ECC Memory 44
8.2 Ranked Memory (and its effect on Linux) 45
8.3 DDR Memory and Dual-Pumped Architectures 46
8.4 Linux Boot Process 48
8.5 BSD vs. GNU Licensing 50
8.6 PowerPC™ DMIPS Benchmarking (Future Work) 51
8.7 The IBM CoreConnect™ Bus 51
8.8 DLL vs. PLL (a.k.a Xilinx vs. Altera) 53
8.9 The XST Flow 53
8.10 Construction of an Ethernet Interface 56
8.11 Serial Communication with the XUPV2P Board 57

9 FUTURE WORK **57**

10 CONCLUSION **58**

11 REFERENCES **58**

APPENDIX **65**

A.1 Minimum Kernel Configuration w/o RFS 65
A.2 Kernel Boot Log w/o RFS 65
A.3 Minimum BusyBox Configuration 66
A.4 Minimum Kernel Configuration w/ RFS 67
A.5 Kernel Boot Log w/ RFS 68
A.6 File Descriptions 69
A.7 Block Diagram 71
A.8 Memory Map 72
A.9 Acronyms 73

LIST OF ILLUSTRATIONS

Figures

FIGURE 1: THE BSB WIZARD 12
FIGURE 2: SETTING PROJECT PATHS 13
FIGURE 3: CHOOSING A BOARD 14
FIGURE 4: SELECTING THE PPC 15
FIGURE 5: PPC CONFIGURATION 16
FIGURE 6: UART16550 CONFIGURATION 17
FIGURE 7: ETHERNET_MAC AND SYSACE CONFIGURATION 18
FIGURE 8: PLB DDR SDRAM CONTROLLER 19
FIGURE 9: PLB BRAM IF CONTROLLER 20
FIGURE 10: STDIN/STDOUT CONFIGURATION 20
FIGURE 11: SUMMARY WINDOW 21
FIGURE 12: SOFTWARE PLATFORM SETTINGS 22
FIGURE 13: OS AND LIBRARY CONFIGURATION 23
FIGURE 14: ENABLING THE BOOT LOOP 24
FIGURE 15: HOW RANKS CAN BE APPLIED TO A DIMM [46] 46
FIGURE 16: IBM CORECONNECT™ BUS ARCHITECTURE [59] 52
FIGURE 17: 8086 (MIN MODE) PIN-OUT [61] 53
FIGURE 18: XILINX DESIGN FLOW (1) [65] 54
FIGURE 19: XILINX DESIGN FLOW (2) 54
FIGURE 20: XILINX DESIGN FLOW (3) 55

Tables

TABLE 1: REQUIRED HARDWARE 9
TABLE 2: REQUIRED SOFTWARE 10
TABLE 3: DDR TYPES 47

1 Disclaimer

If you "let the smoke out" or otherwise ruin the development board while working through this document I take no responsibility - continue at your own risk! Additionally, unless **explicitly** stated, no Linux command shown should be run as '`root`' user.

I have done my best to ensure that all of the information presented is correct; however, I would very much appreciate any feedback or questions. I will be converting this tutorial into HTML and PPT format and will be making them publicly accessible on my website [2]. Please send all e-mail to *webmaster [a t] rm-rfroot.net*.

2 Acknowledgments

Many thanks to my department manager, Patrick L. Manke, at Sandia National Laboratories for providing me with the required development software, and to my fellow co-worker, Diana Eichert, for her infinite help and motivation. I would also like to send my sincere appreciation to Professor Warren R. Carithers [3] for taking time out of his busy schedule to monitor and evaluate this research; to Jamie Lin [4] of Washington State University and to Brian C. Van Essen [5] of the University of Washington for their input during this project. Special thanks to Xilinx Inc. [6] and their technical support staff for designing and maintaining this amazingly versatile board, and making this independent study possible.

3 Introduction

This project is my attempt to learn more about operating systems, computer programming, and hardware design. Open source software and hardware is one of the best learning tools available and this independent study is a great opportunity to exploit that tool to the best of my abilities. I strongly encourage the reader to review the following referenced article published by Xilinx for a good, high-level overview of what will be demonstrated in this document [1].

The *Approach* section is ultimately an in-depth tutorial on the process of porting MontaVista Linux [7] to the XUPV2P development board [8] using the Xilinx EDK and ISE software [9, 10]. Some of the steps in the tutorial or other concepts discussed in the *Lessons Learned* section may seem elementary for some but my goal in creating this document is to make it understandable and useful to as broad an audience as possible.

Additionally, for the sake of organization (and my own sanity) all acronym definitions have been placed at the very end of the *Appendix*. Moreover, I will continue to add more information to this document even after this independent study has been completed. New sections or modifications will be made based on user feedback and/or my own research. I will add an update log to record any changes. Updated versions may be found on my website [2].

4 Objectives

The two major components of this study are to first create an embedded hardware system and then integrate a unique software operating system which utilizes that embedded design in an efficient manner. The computer science aspect is found in building a well-organized software and hardware architecture than can be effectively combined into one user-reliable operating system.

4.1 Primary Objectives

- Learn a new operating system (embedded Linux) and understand cross compiler tool techniques. The OS will be demonstrated utilizing the hard PowerPC cores located within the FPGA.

- Explore various bootstrap methods, including but not limited to Compact Flash and diskless NFS.

- Extend studies from my past computer architecture course, including but not limited to performance analysis, benchmarking (e.g. DMIPS), and effective hardware design from a computer science viewpoint.

- Establish an understanding of the new Xilinx Virtex-II Pro FPGA which I will also be using in my thesis with the Force10 MetaTraffic Processor [11].

- Study an entirely new CPU architecture (PowerPC) and examine how it can be exploited to benefit the hardware and software operations of the device used in my thesis.

- Produce and maintain *reusable* IP cores which can be applied to a wide range of embedded operating systems [12].

- Familiarize myself with the software tools and programming languages that will be used in my career.

- Design a hardware-software interface to allow a user to interact with the dynamic hardware via the open-source OS

4.2 Secondary Objectives

- Experiment with the Xilinx MicroBlaze [13] soft-core processor available to all users of the EDK which can run uClinux.

- Explore the benefits of ECC memory.

- Experiment with the new IBM CBE [14] for PowerPC running on the PowerPC Linux OS.

- Create a communications channel between the two PowerPCs cores and observe the benefits of dual-core operation.

5 Motivation

In August of 2005 the Department of Defense and OMB issued a memorandum [77] stating that all U.S. government agencies must be IPv6 compatible by June 2008. The largest concern is there are very few hardware/software security measures that have been developed to defend against IPv6 intrusions/exploits. A proactive development approach must be taken in order to defend against a protocol that is still in its infancy. This is especially true for 1Gb and 10Gb speeds, which most federal facilities have had in place for quite some time.

MetaNetworks Technologies Inc. [15], now part of Force10 Networks [11], has built highly capable IDS/IPS systems, based on the Snort rule-based, network security language [16], that function perfectly up to the necessary 10Gbps speeds and include support for IPv6. Sandia Laboratories owns a 10Gbps version of this device. With a background in both computer engineering and computer science I jumped at the chance to work with the box knowing that it would be the perfect blend of both curriculums.

Because I am attending college on a fellowship sponsored by my employer, Sandia National Laboratories [17], my Master's thesis must be beneficial to at least one of Sandia's many areas of research. MetaNetworks has agreed to provide me with all of the hardware resources that I will require for developing IPv6 IDS/IPS on their devices. Livio Ricciulli, the founder, president, and chief scientist of MetaNetworks has also agreed to be my mentor throughout the process and chair on my thesis committee.

I have been compiling and reading many RFCs, literature, and the like on Snort, IPv6 and IDS/IPS systems. Livio has also begun sending me literature, manuals, etc on the hardware. I will have access to top-of-the-line networking equipment while developing my thesis and be allowed to work on it full-time at Sandia. Many of my co-workers at the labs are also more than willing to provide any help that I might need on the project. They realize how important it is that this technology be developed and refined as soon as possible. Please visit my thesis website [18] and/or subscribe to the RSS feed for progress updates.

This independent study is the perfect lead into my thesis which I will be continuing research on this year. It is my hope that learning how to utilize the PowerPC cores within the Virtex-II Pro FPGA will allow me to exploit their extra processing power in order to gain efficiency within the MetaNetworks device.

6 Approach (Tutorial)

Before moving onwards, I strongly encourage any person about to read this tutorial to first attempt the embedded Linux port completely on their own. Much more knowledge and understanding of the system will be gained by doing it alone rather than by following step-by-step instructions. My first suggestion for starting any research is (and always will be) Google [19] but, of course, the instruction manuals for the development board are also a good choice. For some quick answers try reviewing the websites in the *References* section of this document.

In the most general case, porting an RTOS to the Xilinx V2P or V4FX development boards consists of the following steps:

1) Acquire the preliminary hardware and software.

2) Run the BIST to verify board functionality.

3) Create a hardware design within the FPGA that will be the foundation of the RTOS.

4) Create a cross-compiler environment with which to build the PowerPC compatible Linux kernel.

5) Build the RTOS kernel for use with the specialized Xilinx hardware and software.

6) Test the kernel boot image.

7) Build an RFS for the kernel to mount at startup.

8) Test the kernel boot image **and** file system together.

6.1 Right Tools for the Right Job

We won't be able to do much of anything unless we have the right tools. Below I have included a table of all hardware and software necessary to complete the aforementioned steps. Earlier versions of some software listed below may be used but might require more effort to get running. Xilinx's development software also fully supports Red Hat Linux [78] and Solaris [79]. The hardware design steps should be quite similar if not exactly the same no matter what operating system is chosen.[1]

Hardware Packages	Retail Cost	Exact Version Used
XUPV2P Board	$1700 ($300)[2]	Revision C
PC (Windows XP) (optional)	$????	Windows XP Service Pack 2, 1.7GHz Pentium M, 2GB RAM
iMacG3 (optional)	$250	G3 500, 512MB RAM
Parallel Cable IV (optional)	$100	N/A
512MB PC2100 SDRAM	$70	Kingston®, Dual Rank, Non-ECC
Miscellaneous Devices	$45	N/A
256MB Compact Flash	$30	SanDisk® Ultra-II 1GB

Table 1: Required Hardware

Software Packages	Retail Cost	Exact Version Used
Xilinx ISE Foundation	$2500	8.1i Service Pack 3

[1] Choosing to run the Xilinx software on Linux will make the entire process much more streamlined as there will be no need to jump back and forth between Windows/Solaris and Linux.
[2] A registered college student can receive the XUPV2P board for the lower price.

Xilinx ChipScope Pro (optional)	$700	8.1i Service Pack 3
Xilinx EDK	$550	8.1i Service Pack 2
MentorGraphics® ModelSim DE/SE (optional)	$2500	6.1e SE
Favorite Linux Distribution	Free!	2.6.12-gentoo-r6 #10 1.7GHz AMD Sempron
MontaVista PowerPC Linux Developer Edition	Free!	2.4.26
Tera Term Pro	Free!	3.1
BusyBox	Free!	1.2.0
Crosstool	Free!	0.42
WinSCP	Free!	3.8.2

Table 2: Required Software

Total cost of resources: $3,845 (w/ educational discount and excluding optional h/w & s/w)

Obviously, this is not a cheap project by any means which is why I suggest the reader have either their school or employer pay for the necessary tools. The earliest version of ISE and EDK that support embedded Linux RTOSs is 6.1i. The CAST [20] department at RIT [21] has access to the necessary Xilinx software (at the time of this writing, version 7.1i) for registered students. Visit the ECTET [22] department office and request access to the ESD lab or stop in the lab during any open lab hours. The following list explains the reason for needing each of the aforementioned resources:

6.1.1 Hardware

- **XUPV2P Development Board** – the main tool used in this project, designed by Xilinx Inc. as a learning tool for students and engineers alike. There are other smaller boards available [23] but this is by far the most versatile of the line-up.

- **PC** – only required if the user intends to run the Xilinx tools on a Windows platform rather than Linux. Running everything on Linux will make the entire process much faster as there will be no need to transfer files back and forth between environments.

- **iMacG3** – I recently purchased this to learn how PowerPC Linux operates in a non-experimental environment. It can also be used for verification purposes once PowerPC Linux is up and running on the XUPV2P board.

- **Parallel Cable IV** – used for programming the PROM device which uploads the design bit stream and small software applications to the FPGA and BRAM, respectively, at power-on. It can also be used to attach to the CPU debug port. A USB cable may also be used which is much faster than the PCIV – and much cheaper.

- **512MB PC2100 SDRAM** – This will be used as the embedded system's main memory. I have obtained both Error Correction Code (ECC) and non-ECC memory for possible fulfillment of the secondary objective listed above. Non-ECC PC2100 memory is

approximately half the cost of its ECC counterpart and *much* easier to find. The version that I used is available directly from Digilent's website [23]. A 256MB module can also be used – it depends on how much space the user wants to have available as a development "playground".

- **Miscellaneous Devices** – VGA cable for frame buffer output, Serial cable for UART, USB for PROM/FPGA programming, Ethernet cable for network testing, SATA cable for testing, and CF card reader for file transfer.

- **512MB Compact Flash** – used to store the kernel, RFS, and swap partitions for load at boot-time. The swap partition should be at least as large as the amount of main memory available so a smaller capacity is not recommended. If much more space is needed then I would suggest purchasing an IBM MicroDrive™ or even a Seagate® ST1. Seagate produces a 12GB CF drive with 2MB of cache memory.

6.1.2 Software

- **ISE Foundation** – the core development environment needed to interface with the FPGA hardware. This software will take the VHDL, Verilog, schematic, etc. information and generate a programming file which can be written to the onboard PROM

- **ChipScope Pro** – can be used as a "virtual" logic analyzer internal to the FPGA. The ISE software can instantiate a ChipScope Pro module within the FPGA which can be used for verification and debugging purposes.

- **EDK** – this is the Xilinx Embedded Development Kit and Platform Studio software which is the only way to interface with the PowerPC cores within the Virtex-II FPGA

- **ModelSim** – this is a debugging and simulation tool for VHDL and Verilog which can be used as an extension of the ISE software. Xilinx can directly interface with the ModelSim software which allows for a very nice development combination.

- **Favorite Linux Distribution** – used for all software development

- **MontaVista Linux** – the open source OS that will be used

- **Tera Term Pro** – used to interface with board over UART16550

- **BusyBox** – used to generate root file system

- **Crosstool** – used to create the cross-compiler tool-chain needed for developing software for the PowerPC processor.

- **WinSCP** – used to quickly and easily transfer files from Windows to Linux (only needed if running Xilinx tools on Windows machine)

6.2 Testing the Board

The first step in working with a test board, which has been provided by another company or department, is to ensure that it is fully functional. No one wants to troubleshoot issues that were not their fault to begin with. For this reason, Xilinx has provided their customers with a BIST environment which comes pre-programmed into the PROM.

Even though the BIST does not perform a completely exhaustive test of the board and all of the available IP cores, it does come quite close. Xilinx has included great documentation to accompany the BIST (in the '`docs`' folder) which will get things moving along. The BIST reference design can be downloaded from the Xilinx [8] or Digilent [24] websites.

Testing of the board also includes at least a limited understanding of its capabilities. At a minimum, read the XUPV2P's user guide [25]. There is no sense in continuing with this tutorial without a moderate knowledge of how to work with the hardware and how it is configured. The user's guide will also explain how to restore the BIST should it accidentally be overwritten in the PROM.

6.3 Creating a Solid Foundation

With the base verification phase complete we will need to generate a reliable hardware foundation on which to develop our embedded Linux OS. The software design aspect can only be as good and as capable as the hardware which executes it. Hence, it is essential that we perform this step correctly the first time.

In a practical business setting it is highly unlikely that a design's initial revision would be without flaw. The hardware would be sent back for numerous changes during a project's life cycle. But for this endeavor we will have the help of Xilinx's amazing design and debug tool suite.

6.3.1 Configuring the PowerPC Hard Core Processor

Before going any further please note that if any version of EDK other than 8.1i is being used the GUI is quite different. To create the hardware design for this project we will be using the Xilinx EDK BSB Wizard. Launch EDK and select the Base System Builder Wizard.

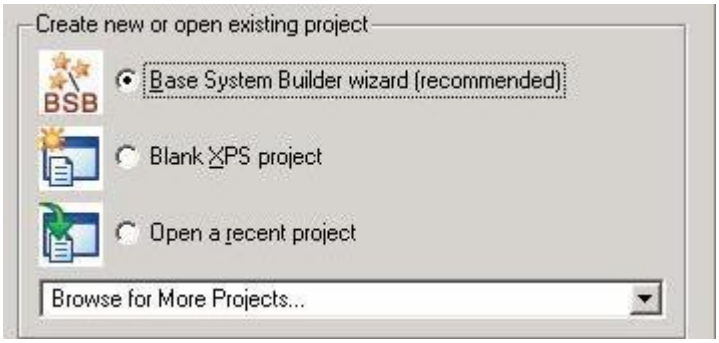


Figure 1: The BSB Wizard

In the next screen select the destination for the project directory and ensure that the location does not include any spaces and does not reside on a network drive. Also select the repository path which includes the necessary IP cores and the XUPV2P board definition file. The IP cores include the software and hardware source files for the SDRAM memory controller, Ethernet controller, RS-232 port, etc.. The board definition file provides the BSB wizard with specific information about the XUPV2P board's design and layout. The repository can be downloaded at the Digilent website [26].

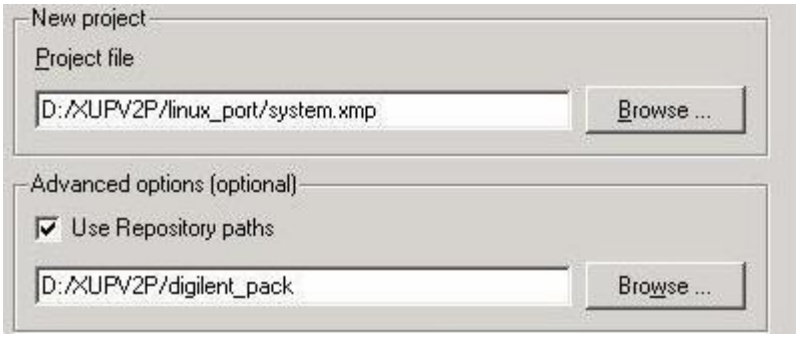


Figure 2: Setting Project Paths

On the following screen choose "**I would like to create a new design**" and click Next. Choose the board vendor, name, and revision as shown in Figure 3. The information being selected in the boxes (including all of the peripheral options in the screens to come) is located in the board definition file provided by Digilent.

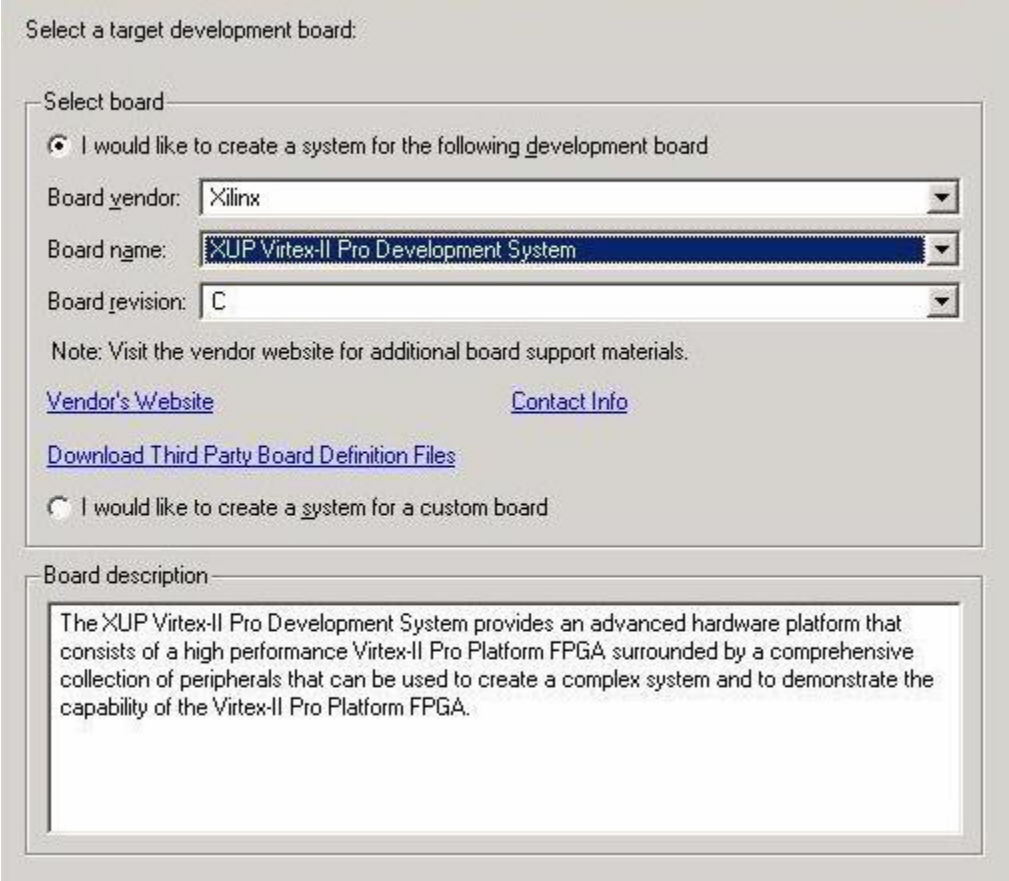

Select a target development board:
Select board
I would like to create a system for the following development board
Board vendor: Xilinx
Board name: XUP Virtex-II Pro Development System
Board revision: C
Note: Visit the vendor website for additional board support materials.
Vendor's Website
Contact Info
Download Third Party Board Definition Files
I would like to create a system for a custom board
Board description
The XUP Virtex-II Pro Development System provides an advanced hardware platform that consists of a high performance Virtex-II Pro Platform FPGA surrounded by a comprehensive collection of peripherals that can be used to create a complex system and to demonstrate the capability of the Virtex-II Pro Platform FPGA.

Figure 3: Choosing a Board

On the next screen we will choose the hard core PowerPC processor rather than the soft core MicroBlaze processor. It is important to note that there are some embedded operating systems that support Xilinx's soft core MicroBlaze processor [27] but we will not be using them at this point. Understand that instantiating MicroBlaze will require a significant number of resources inside the FPGA whereas the PPC405GP requires no extra resources whatsoever since it is *immersed* in the FPGA fabric.

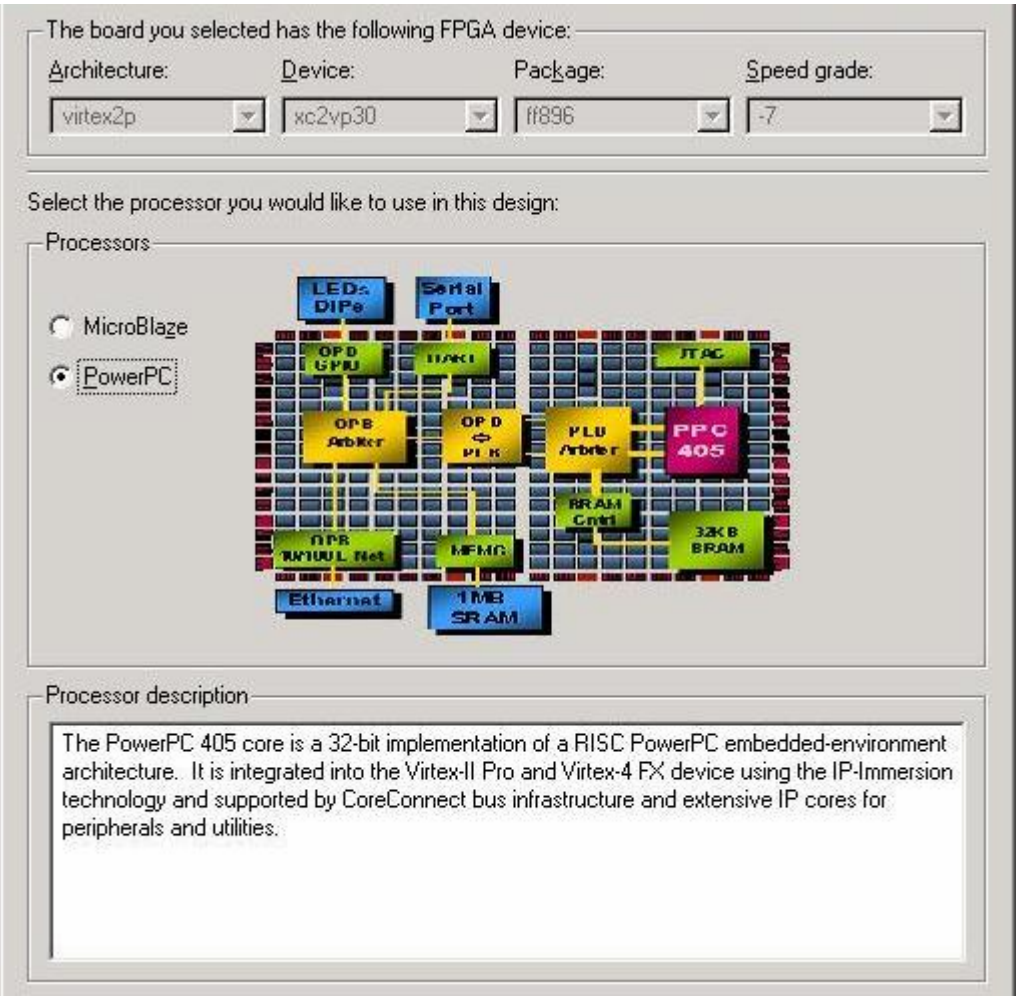


Figure 4: Selecting the PPC

The only thing to be changed in Figure 4 is the PPC's operating frequency to 300.00MHz, but there is a lot of information here that needs to be discussed. First off, the PPC405 can operate at up to 400MHz but we see that there is no option for that speed. Any speed above 300MHz must be entered into the UCF manually. The reason for this is there are special timing constraints that must be taken into consideration if the PPC is to run at that top speed. If Xilinx allowed 400MHz to be selected in the wizard it is quite likely that the designer would forget to set the other appropriate parameters and spend much unnecessary time debugging.

The bus clock frequency (a.k.a reference clock and labeled as part 'Y2' on the PCB) is utilized by the DCMs inside the FPGA. The DCMs can multiply or divide an input clock's frequency to suit the needs of an internal component. In this case, one of the DCMs is used to multiply the reference clock by 3 for the PPC405. See the *Lessons Learned* section for more information about DCM.

The bus clock is also referenced by the DDR SDRAM via another DCM which is used to bring the clock to the proper speed for PC2100 DDR specifications (i.e. 66MHz). It is important to remember, however, that DDR memory is "double-pumped" (i.e. data is sampled on both the

rising and falling edges of the clock) off of this reference clock thereby creating a 133MHz sampling rate.

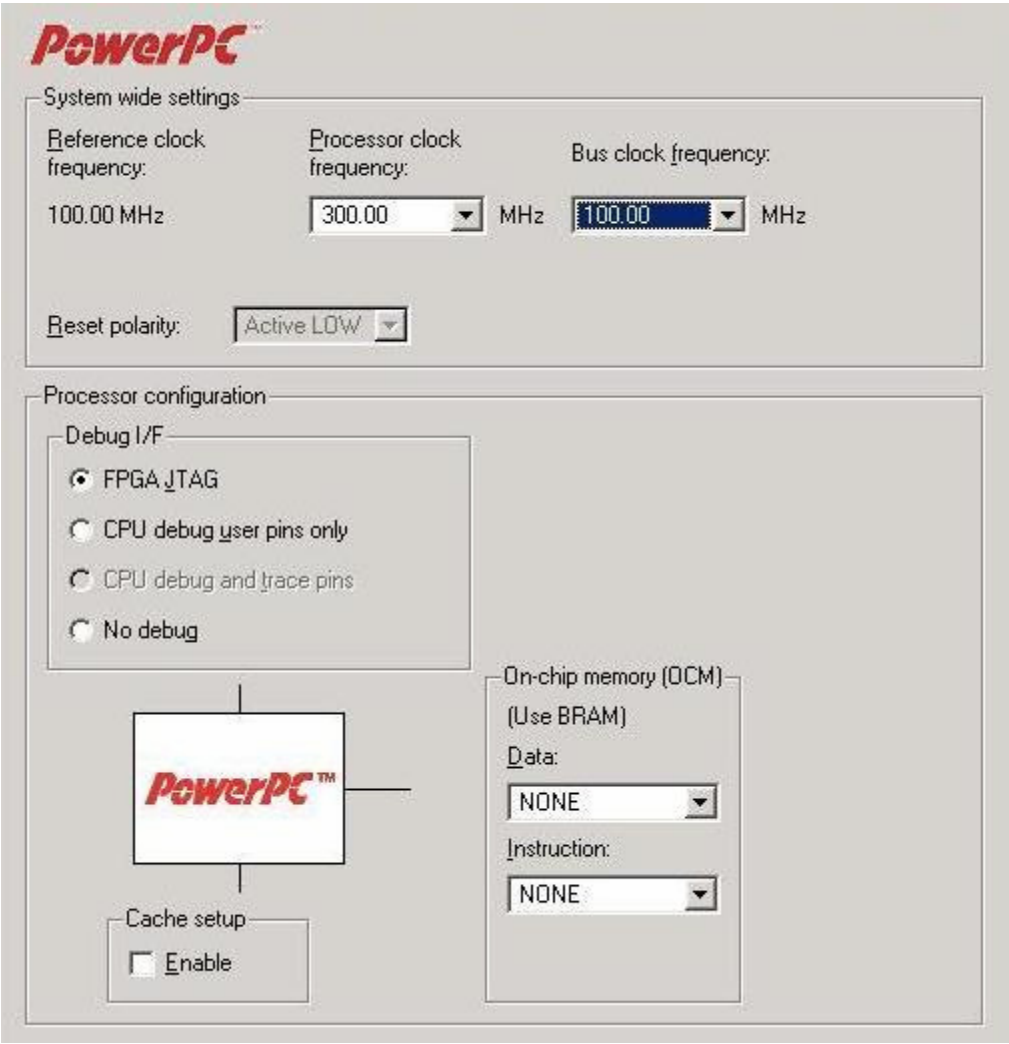


Figure 5: PPC Configuration

6.3.2 Configuring the Peripherals

We will now select and configure the peripherals that will be available to the Linux kernel and the user. During this part of the tutorial be sure to read the data sheets for each of the peripherals that we will be using. The information they provide will be extremely important later when we need to use each peripherals API.

Figure 6 and Figure 7 show the UART16550[3], Ethernet_MAC, and SystemACE configurations. We have chosen to connect these peripherals to the Xilinx OPB interrupt controller core (rather

[3] MontaVista also supports the Xilinx OPB UARTLITE which is fixed at 9600 baud. If slower output speed is not an issue then I suggest using the evaluation version instead. A non-evaluation version of the Ethernet controller is not available, however, I may be creating one for my thesis to better understand packet parsing. It may or may not be released to the public domain as it is ultimately up to my employer.

than using software interrupts) for faster performance. The OPB is for low-speed, low-utilization devices. The PLB is for high-speed, high-performance devices such as the DDR memory. Please refer to the *Lessons Learned* section for more information on the PLB and OPB which are part of the IBM CoreConnect™ Bus architecture.[4]

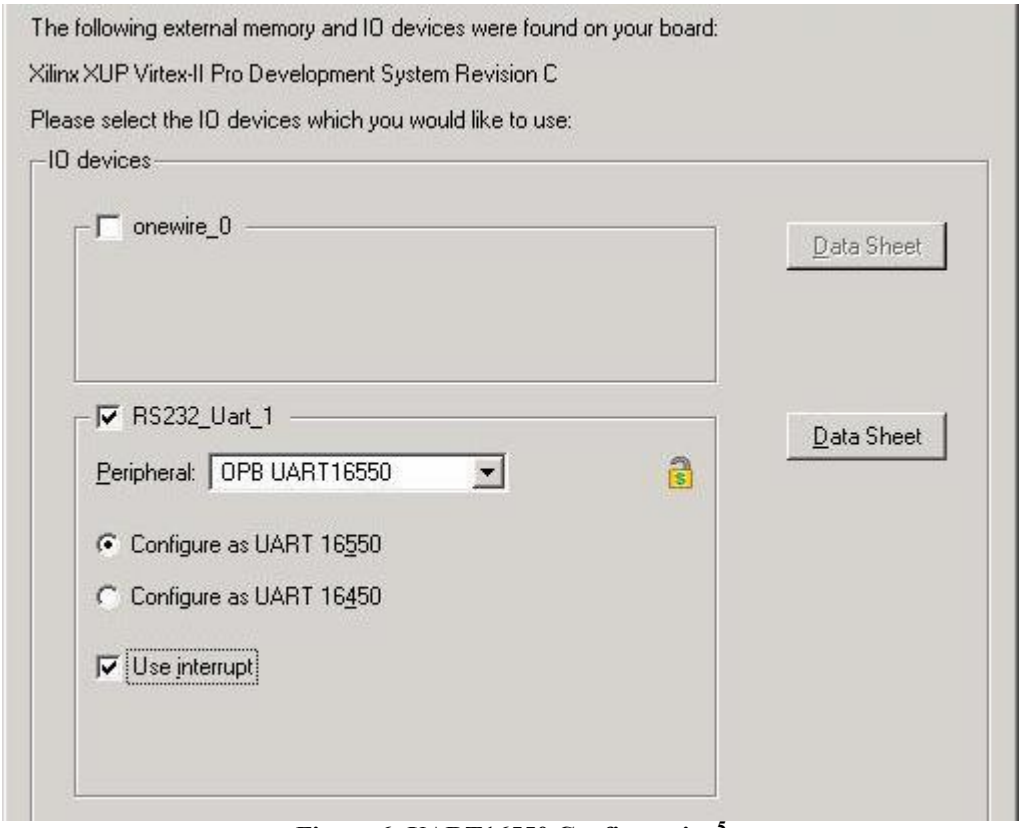

The following external memory and IO devices were found on your board:
Xilinx XUP Virtex-II Pro Development System Revision C
Please select the IO devices which you would like to use:
IO devices
onewire_0
Data Sheet
RS232_Uart_1
Data Sheet
Peripheral:
OPB UART16550
Configure as UART 16550
Configure as UART 16450
Use interrupt

Figure 6: UART16550 Configuration[5]

[4] The Ethernet_MAC controller core does not have to be added if the reader does not intend to add networking support to Linux. I included it here because it's easier to add it with the wizard interface than to add it manually later. The Ethernet_MAC core is discussed later in this tutorial when networking support is added to the board.

[5] The number of peripheral devices displayed on each wizard page will depend on the resolution of the monitor.

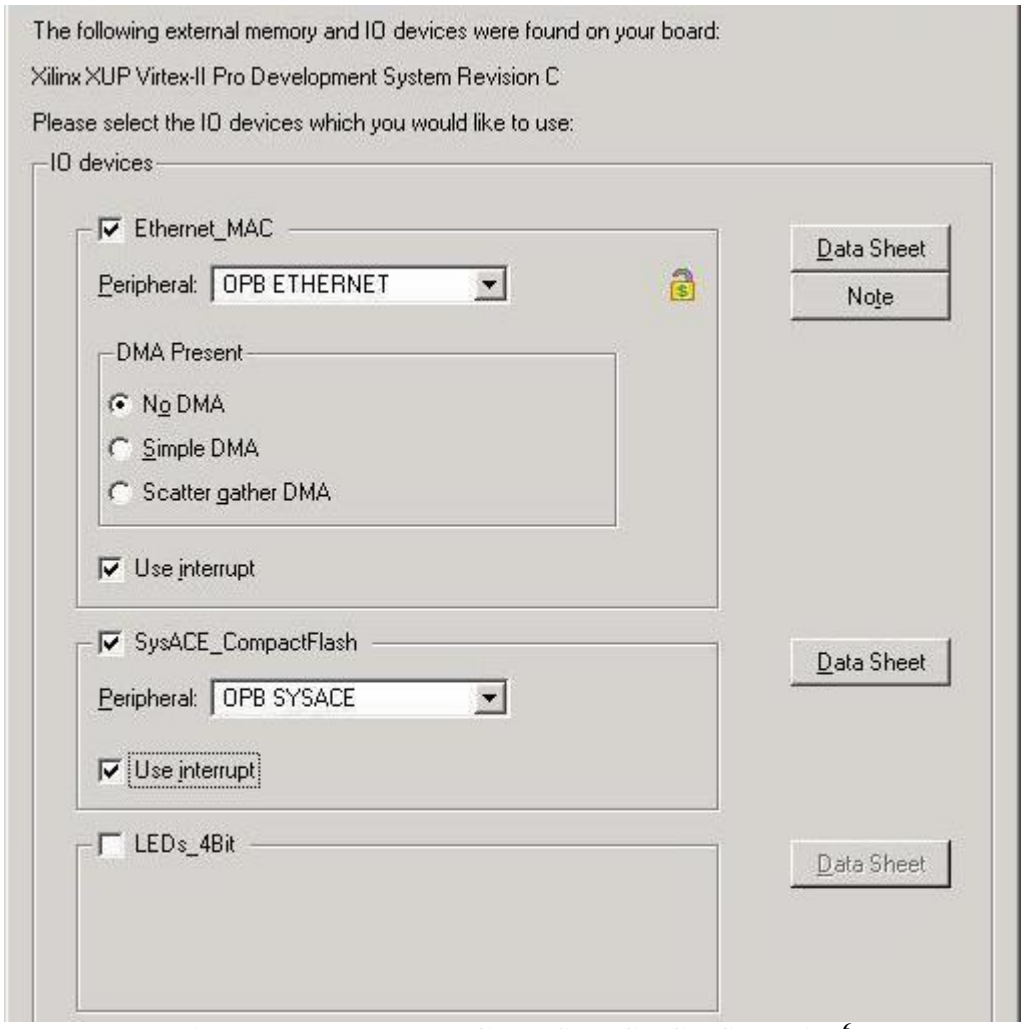


Figure 7: Ethernet_MAC and SysACE Configuration[6]

Figure 8 shows the external DDR memory configuration. Choose whatever memory size and rank matches the memory module purchased. The interrupt option for the PLB DDR controller may seem strange at first since one would think of local memory as a passive system component. This is true unless the memory module uses ECC capable and the design will actually be taking advantage of that capability.

Note, however, that the Xilinx PLB DDR controller only supports ECC for 32-bit SDRAM modules. 32-bit SDRAM modules are a different form factor than 64-bit modules and are not compatible with the XUPV2P board. If an ECC-compatible, 64-bit DDR controller is required here are a few options:

1) Buy one from a vendor
2) Find one online [74]

[6] The lock icon next to some of the peripherals is there to inform the designer that this IP core must be purchased from Xilinx to use them in a production design (of course, one could always create their own). The IP core will still function in the FPGA but only for 8 hours. At that time, the bitstream will have to be uploaded to the FPGA again.

3) Modify Xilinx's controller to support 64-bit
4) Write one own from scratch

If the interrupt line is enabled by mistake when the system is built, and the RAM is not actually ECC-compatible, the design will not work since the DDR controller will attempt to read and write to the CB lines on the DIMM which are not connected to anything and also marked as DNU in the module's datasheet. See *Lessons Learned* for more information on ECC memory.

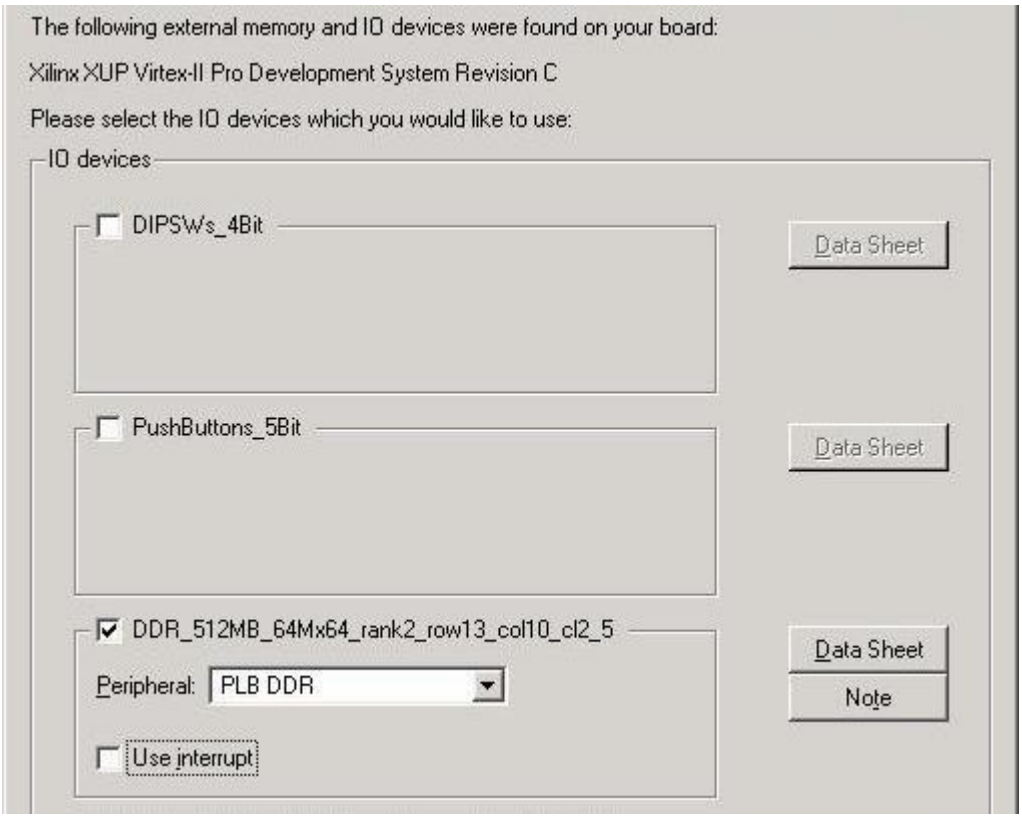


Figure 8: PLB DDR SDRAM Controller

All remaining peripherals on any other wizard pages should be disabled. We will not be using the PS2, VGA frame buffer, or AC 97 audio as yet – I am leaving that for another more advanced tutorial. For our system, all user and OS input/output will be conducted across the serial port (UART16550) using Tera Term Pro.

Figure 9 and Figure 10 show the XUPV2P board PLB BRAM IF controller and STDIN/STDOUT configurations, respectively. The PLB BRAM IF controller supports up to 128KB of BRAM. There is a total of 306KB of BRAM on the V2P [28] leaving 178KB of internal BRAM free (e.g. for use with custom user applications). STDIN and STDOUT will be defined as the RS232 UART. We will not need any of the sample applications provided by Xilinx.

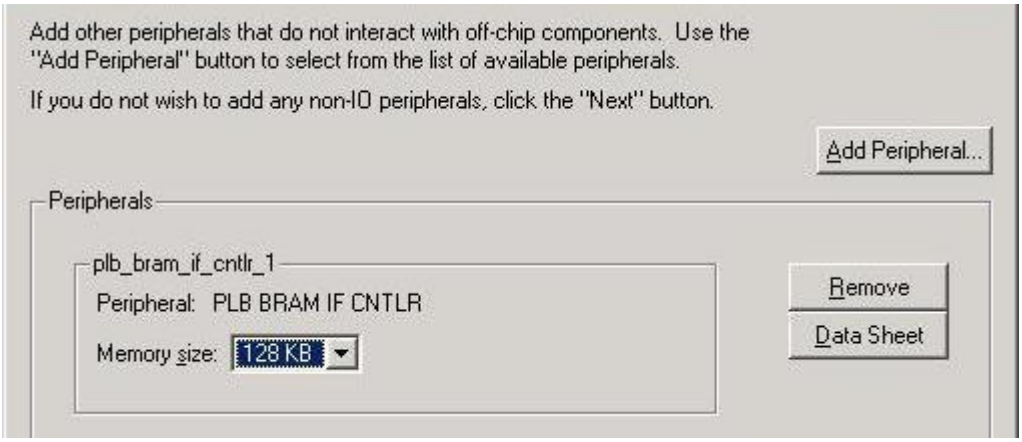

Add other peripherals that do not interact with off-chip components. Use the "Add Peripheral" button to select from the list of available peripherals.
If you do not wish to add any non-IO peripherals, click the "Next" button.
Add Peripheral...
Peripherals
plb_bram_if_cntlr_1
Peripheral: PLB BRAM IF CNTLR
Memory size: 128 KB
Remove
Data Sheet

Figure 9: PLB BRAM IF Controller

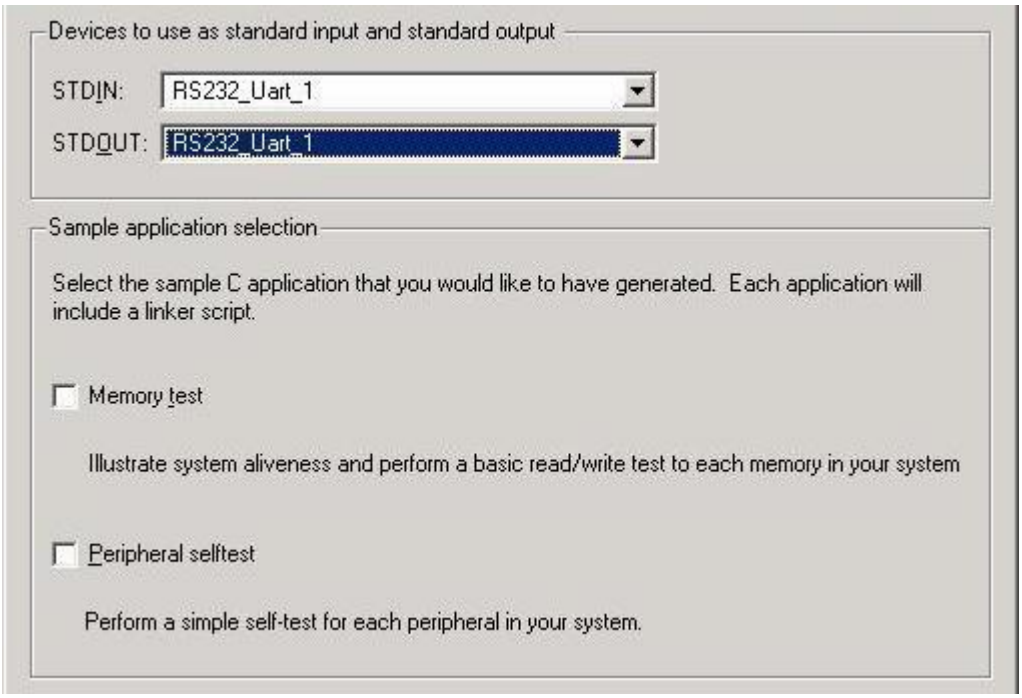

Devices to use as standard input and standard output
STDIN: RS232_Uart_1
STDOUT: RS232_Uart_1
Sample application selection
Select the sample C application that you would like to have generated. Each application will include a linker script.
Memory test
Illustrate system aliveness and perform a basic read/write test to each memory in your system
Peripheral selftest
Perform a simple self-test for each peripheral in your system.

Figure 10: STDIN/STDOUT Configuration

The final page shown in Figure 11 provides a summary of the entire system. If the 512MB **dual rank** memory module was chosen for the PLB DDR controller, notice (in the upper window) that XPS broke each rank into two separate 256MB locations. We will come back to this later as it can cause issues when running our embedded Linux system.[7]

[7] It is a common misconception that if there are DRAM chips physically located on both sides of the memory stick's PCB it is dual rank memory. This is simply untrue as it could still be a single rank device (see Figure 15). Be sure to check with the manufacturer or the data sheet to determine the specific type of module. Check the *Lessons Learned* section of this document for more information on dual rank memory.

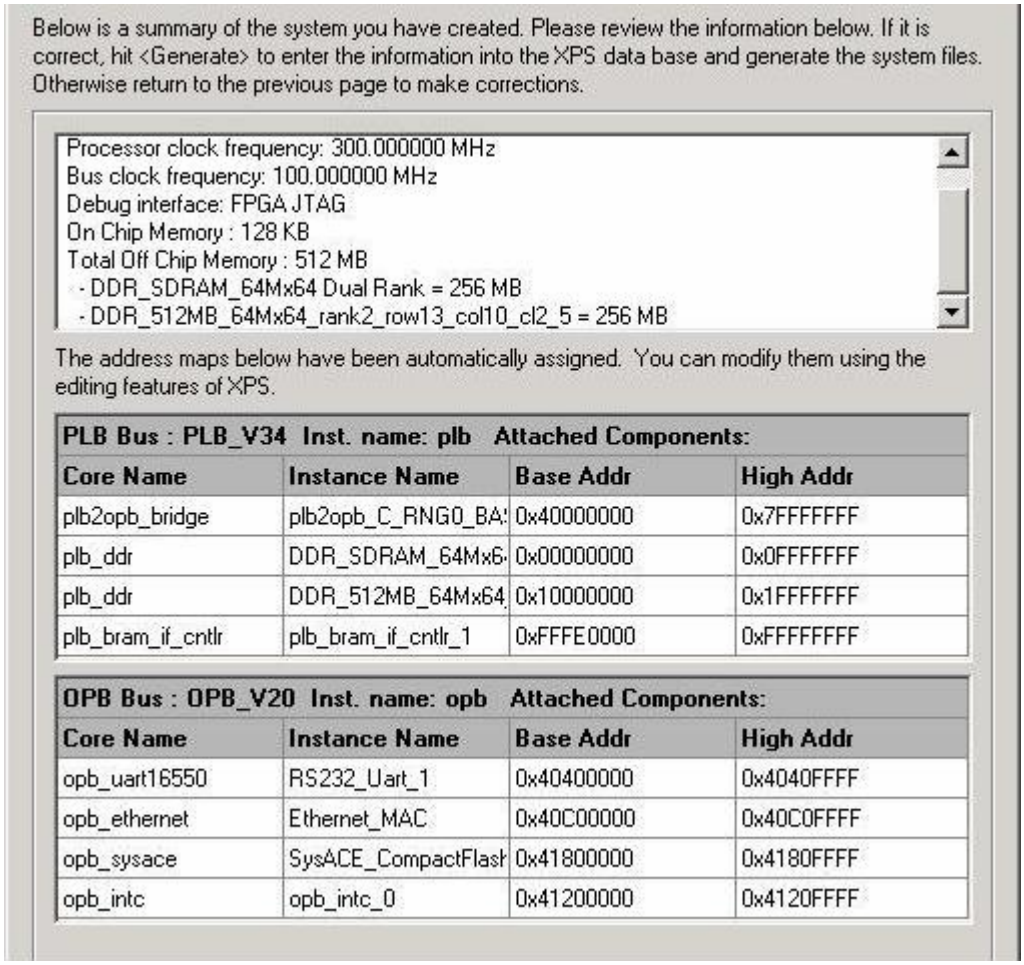

Below is a summary of the system you have created. Please review the information below. If it is correct, hit <Generate> to enter the information into the XPS data base and generate the system files. Otherwise return to the previous page to make corrections.
Processor clock frequency: 300.000000 MHz
Bus clock frequency: 100.000000 MHz
Debug interface: FPGA JTAG
On Chip Memory : 128 KB
Total Off Chip Memory : 512 MB
- DDR_SDRAM_64Mx64 Dual Rank = 256 MB
- DDR_512MB_64Mx64_rank2_row13_col10_cl2_5 = 256 MB
The address maps below have been automatically assigned. You can modify them using the editing features of XPS.
PLB Bus : PLB_V34 Inst. name: plb Attached Components:
Core Name | Instance Name | Base Addr | High Addr
plb2opb_bridge | plb2opb_C_RNG0_BA | 0x40000000 | 0x7FFFFFFF
plb_ddr | DDR_SDRAM_64Mx6 | 0x00000000 | 0x0FFFFFFF
plb_ddr | DDR_512MB_64Mx64 | 0x10000000 | 0x1FFFFFFF
plb_bram_if_cntlr | plb_bram_if_cntlr_1 | 0xFFFE0000 | 0xFFFFFFFF
OPB Bus : OPB_V20 Inst. name: opb Attached Components:
Core Name | Instance Name | Base Addr | High Addr
opb_uart16550 | RS232_Uart_1 | 0x40400000 | 0x4040FFFF
opb_ethernet | Ethernet_MAC | 0x40C00000 | 0x40C0FFFF
opb_sysace | SysACE_CompactFlash | 0x41800000 | 0x4180FFFF
opb_intc | opb_intc_0 | 0x41200000 | 0x4120FFFF

Figure 11: Summary Window

Now that we have the hardware configuration summary we can manually generate a graphical address map. The hardware address map is by far one of the most important things to know when working with a new system – without it we can't do much of anything. A visualization of the final address map is provided in *Appendix* section A.8.

Congratulations! We have now finished defining our base hardware system on which Linux will be running. Click **Generate** and then click **Finish**. For a final block diagram of our system's architecture please see *Appendix* section A.7.

6.3.3 Configuring the Software Platform

The XPS main window will appear following the hardware configuration summary. The interface may seem quite intimidating and complex at first, but this tutorial is meant to remedy that issue. For the open source fanatics reading this document, it may be of interest to note that the XPS GUI is based on the Eclipse open source IDE [80]. Also take note of the extra console window that runs in the background. This window, and the Console Log, will provide critical WARNING and ERROR messages regarding the hardware/software design process.

Remember, we have only told Xilinx what we *would like* it to create for us - we haven't actually instantiated any part of the design yet. Before we do that though, let's give XPS a little more specific information about the ultimate purpose of our design, which is to run the MontaVista eOS. Click on the **Software** menu at the top and select "**Software Platform Settings…"**.

Under the **Software Platform** section (Figure 12) change:

- OS to **linux_mvl31** version **1.01.b**

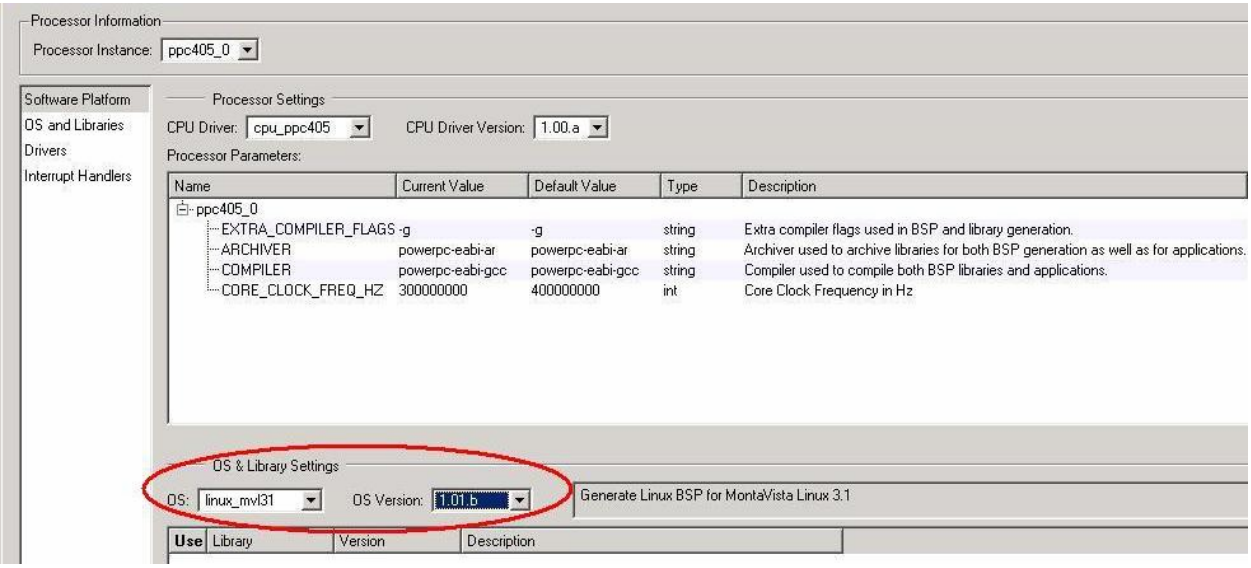


Figure 12: Software Platform Settings

Under the **OS and Libraries** section (Figure 13) set:

- **MEM_SIZE** to **0x10000000**[8]
- **PLB_CLOCK_FREQ_HZ** to 100000000 (100MHz)
- **TARGET_DIR** to a location for storing the BSP
- **Connected_periphs** to add our 4 peripherals: **RS232_Uart_1**, **Ethernet_MAC**, **SysACE_CompactFlash**, **opb_intc_0**

[8] If using more than 256MB of SDRAM please see the *Lessons Learned* section on

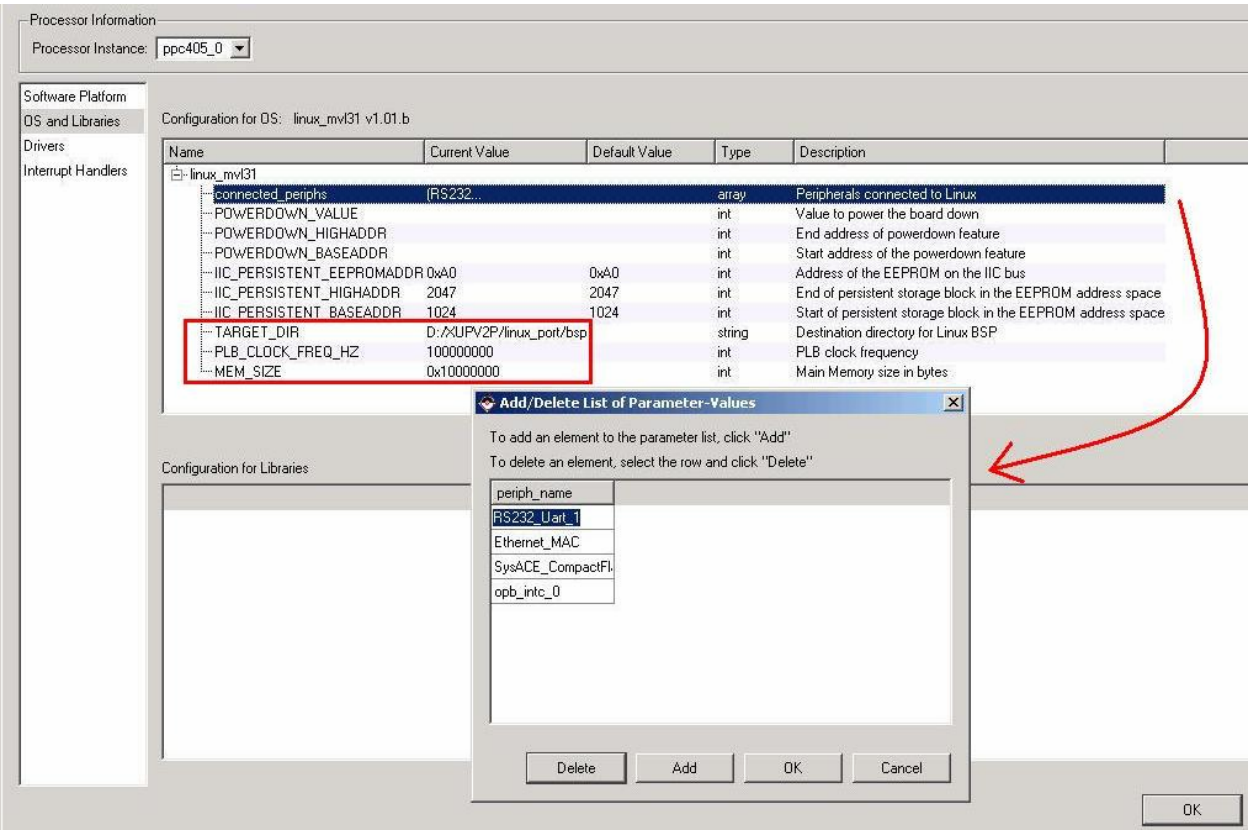


Figure 13: OS and Library Configuration

Click **OK** to any open child windows and then click on the **Applications** tab in the main XPS window and verify that the PPC405_0_bootloop is set to be loaded into BRAM (Figure 14). See the *Lessons Learned* and *Trials and Tribulations* sections for more information on the Xilinx boot loop.

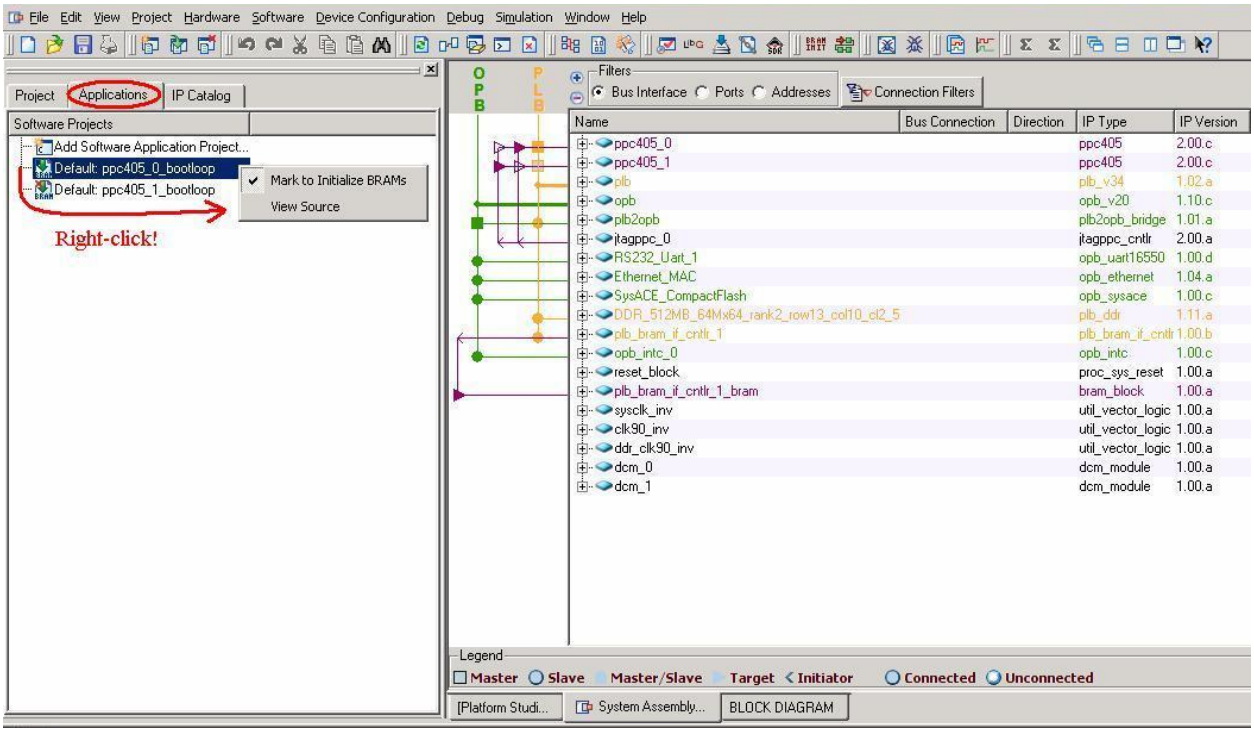


Figure 14: Enabling the Boot Loop

We are finally ready to start building something useful. Select the **Hardware** menu and click "**Generate Bitstream**". This will generate the complete Netlist (a.k.a PlatGen) for our design and also create the final bitstream that will be uploaded to the FPGA.[9] Please refer to *Lessons Learned* for more information on the Xilinx XST flow.

Numerous WARNINGs regarding the generation of a "Black Box" [29] will appear in the Console Log window. Xilinx will generate a "black box" whenever there is no VHDL code associated with a hardware component. These components include: EDIF files, primitives, and CORE [30] modules. More information on this warning can be found on the Xilinx technical support website or by clicking on the link that the console log provides.

Now that we have built our hardware we need to create the software Libraries and BSP for MontaVista to use in the kernel compilation process. The BSP consists of various specialized drivers, header files, device ID numbers, and the system address map. The most important of the generated files is arguably the `xparameters.h` file. This file is essential for allowing the kernel software to interface with any instantiated devices. I suggest reviewing this file and understanding its contents.

To create the BSP, select the **Software** menu and click on "**Generate Libraries and BSPs**" (a.k.a. LibGen). All of the necessary files for MontaVista will be created and stored in the directory specified by the TARGET_DIR variable set earlier.

[9] Two WARNINGs regarding the UART16550 and the Ethernet controller will appear in the Console Log. These are nothing to be alarmed by as Xilinx is stating that those IP cores are for evaluation purposes only.

6.4 Creating the Cross-Compiler Environment

At this point Xilinx has pretty much done all of the automation it can do for us. We now need to create the cross-compiler environment that will be used to compile the PPC405 kernel and Linux RFS. To accomplish this I have chosen to use Crosstool [31] as my research has found it to be a very reliable and easy-to-use tool-chain generator. Crosstool is a set of scripts to build different versions of GCC and GLibc (a.k.a. libc6) written by Dan Kegel.

I have placed a modified Crosstool tarball on my independent study website [2] for anyone wanting to use that version instead. Both are the same software but I have removed all of the unnecessary architecture support (e.g. alpha, x86, MIPS) and made all of the necessary changes. To truly learn how the tools work, however, I suggest making all of the changes manually.

Once Crosstool is downloaded, extract it to '`/home/user_name`' and run the following commands[10]:

```
$ mkdir xupv2p
$ cd xupv2p
$ wget http://kegel.com/crosstool/crosstool-0.42.tar.gz
$ tar -xvzf crosstool-0.42.tar.gz
$ cd crosstool-0.42
$ mkdir downloads
$ sudo mkdir /opt/crosstool
$ sudo chown <user_name>:users /opt/crosstool/
$ vi demo-ppc405.sh
```

Replace: `TARBALLS_DIR=$HOME/downloads`
with: `TARBALLS_DIR=$HOME/xupv2p/crosstool-0.42/downloads`

Replace: `eval `cat powerpc-405.dat gcc-3.4.1-glibc-2.3.3.dat`  sh all.sh –notest`
with: `eval `cat powerpc-405.dat gcc-3.4.**4**-glibc-2.3.3.dat`  sh all.sh –notest`

And add the following lines **above** the '*eval*…' statement:

```
export TARGET=powerpc-405-linux-gnu
export PREFIX=${RESULT_TOP}/gcc-3.4.4-glibc-2.3.3/${TARGET}
export PATH=${PREFIX}/bin:${PATH}
```

We will be using GCC version 3.4.4 because it is the same version used by the cross-compiler in XPS [32]. If 3.4.4 is compatible by Xilinx standards then we will assume it should also be good enough for this development project.

Now would be a great time to run '`screen`' [33] for Linux. If the reader is unfamiliar with screen or have never gotten around to using it, then I suggest changing that fact now. Screen is by far one of the best applications available and is extremely simply to use. With screen we can

[10] These commands will setup the system for non-root user operation. Remember, do **not** run any commands as '`root`' – it is **dangerous**!

keep a constant eye on progress in a different screen session by using the '`less`' command on the log file and hitting *Shift+G*. Or we can simply keep running '`ls | grep *.log`' and make sure the log file size keeps growing.

In any case, run the following command to start the tool-chain compilation process:

```
$ sh demo-ppc405.sh > demo-ppc405.log 2>&1
```

Running the above command will redirect all output to a log file. If there are any errors in the script process simply check the '`demo-ppc405.log`' file. While the interested scholar is waiting, now would be a good time to look through the Crosstool scripts to see exactly how it works. The compilation process took approximately 47 minutes on a 1.7GHz AMD Sempron with 512MB of RAM and consumed 178MB of hard disk space.[11]

6.5 Building the RTOS kernel

Now that the proper tool-chain has been created it can be used to compile the RTOS kernel. Before we can compile the kernel though, it must be configured with the proper parameters for the XUPV2P board and its peripherals. For this demonstration we will be using MontaVista Embedded Linux 3.1 running Linux kernel **2.4.26**.[12] MontaVista was chosen because it supports specific Xilinx hardware including the Xilinx SystemACE Controller, the Ethernet MAC, and the GPIO.

First, download the development branch of the MontaVista Linux 2.4 kernel from their website using '`rsync`' [34] or '`bitmover`' [35]:

```
$ rsync -avz --delete source.mvista.com::linuxppc_2_4_devel
mvistappc_2_4_devel
$ cd mvistappc_2_4_devel
```

Now that we have the kernel we will need to make some modifications to suit our board. After all, MontaVista was designed to run on virtually any embedded system not just the XUPV2P. We will copy the Xilinx XUPV2P BSP (which we created earlier) to our kernel source tree. Copy the '`arch`' and '`drivers`' directories inside of the TARGET_DIR location (that was defined in the previous section) to its counterpart in the mvistappc_2_4_devel directory. Run the following commands if "Permission Denied" errors occur:

```
$ chmod 644 arch/ppc/platforms/xilinx_ocp/*
$ chmod 644 drivers/block/xilinx_sysace/*
$ chmod 644 drivers/net/xilinx_enet/*
```

We now need to modify one of the kernel's Makefiles as it has an error – it attempts to move the resulting kernel image twice and spits out a FNF message. Change the following line in '`arch/ppc/boot/simple/Makefile`' so that it performs a copy rather than a move:

[11] The total time stated does not include the time required to download the necessary source files (also part of the script).

[12] To run Linux kernel 2.**6**.X a different RTOS will likely be desired since MontaVista only provides open source for version 2.4.X. A subscription license must be purchased for anything beyond 2.4.X.

```
zImage-EMBEDDED: zvmlinux
     cp zvmlinux ../images/zImage.$(END)
```

In addition, the Xilinx BSP provides us with new OCP FIFO drivers for the system bus. If we don't use the new versions we will not be able to compile the new Xilinx Ethernet MAC driver that accompanies XPS 8.1i. We can make efficient use of the new drivers by changing only two files. First, make the following change to '`arch/ppc/platforms/xilinx_ocp/Makefile`':

```
xilinx_ocp-objs += xbasic_types.o xdma_channel.o xdma_channel_sg.o \
                   xipif_v1_23_b.o xpacket_fifo_v2_00_a.o \
                   xpacket_fifo_l_v2_00_a.o xversion.o
```

Second, make the following change to '`arch/ppc/platforms/xilinx_ocp/xilinx_syms.c`' in the most obvious place:

```
#include "xpacket_fifo_v2_00_a.h"
EXPORT_SYMBOL(XPacketFifoV200a_Initialize);
EXPORT_SYMBOL(XPacketFifoV200a_Read);
EXPORT_SYMBOL(XPacketFifoV200a_SelfTest);
EXPORT_SYMBOL(XPacketFifoV200a_Write);
```

The change above is the answer to many "UNDEFINED REFERENCE" errors that I encountered. Please refer to the *Trials and Tribulations* section for more information.

For the final modification, and to save those following along some grief, comment out the following line in '`arch/ppc/boot/simple/embed_config.c`':

```
//#error I2C needed for obtaining the Ethernet MAC address
```

We need to do this is because, when the Xilinx-ML300 was chosen as our base system in the kernel configuration, the kernel assumes that there is an I2C SEEPROM used to store the Ethernet MAC address. On the XUPV2P board we are not so fortunate and must hardcode the MAC address prior to the kernel compilation process. I will discuss how to manually configure the MAC address when networking support is added to the board in a later section.

If the shell has been logged out of recently (and the appropriate paths have still not been added to the shell's login script), re-export the proper paths to the PPC405 cross-compiler (the reader's paths may differ than what is shown below):

```
$ export TARGET=powerpc-405-linux-gnu
$ export PREFIX=/opt/crosstool/gcc-3.4.4-glibc-2.3.3/${TARGET}
$ export PATH=${PREFIX}/bin:${PATH}
```

Enable the minimum kernel options with '`$ make menuconfig`' as shown in *Appendix* section A.1. The RTOS kernel will then be ready for compiling:

```
$ make dep && make zImage
```

The compilation process for the minimum kernel configuration took less than one minute on a 1.7GHz AMD Sempron with 512MB of memory. The resulting kernel boot image will be located in '`arch/ppc/boot/images/zImage.elf`'.

I have placed a slimmed-down (i.e. removed unnecessary architecture support), properly configured kernel source tarball (along with the compiled kernel image and BSP) on my website [2]. If hard disk space is an issue, the difference in uncompressed hard disk space of my version is 311MB.

Feel free to use my kernel, but more can be learned from manually modifying and reading the available `HELP` in order to understand why each option was chosen. The `HELP` information contains copious amounts of interesting, useful, and usually **necessary** information.

6.6 Testing the Kernel (w/o RFS)

Now that we have our kernel it is probably best to test it to see if it will boot before we even bother with creating an RFS. Attempting to mount an RFS on top of a botched kernel makes little sense and it might cause confusion as to which part of the design is causing the issue. Professor Carithers and I like to remember these two rules when developing/debugging (in order of importance):

1) "Only change one thing at a time."
2) "Simplify things as much as possible."

With preaching aside, let's create the ACE file which will be used to program the FPGA with the hardware design and load the kernel image. Please refer to the *Lessons Learned* section for more information on the Xilinx SystemACE architecture.

First, create the following directories (or something similar) in the Xilinx EDK project directory (e.g. "`D:\XUPV2P\linux_port\`"):

```
montaVista_mkdosfs (if using mkdosfs under Windows)
montaVista_ACE
montaVista_ELF
montaVista_GenACE
```

Whether running Windows or Linux, find the '`mkdosfs`' utility on the WWW and install it in the appropriate folder. If running Gentoo, simply execute:

```
$ sudo emerge dosfstools
```

Create the file '`montaVista_GenACE/xupGenACE.opt`' and add the following lines:

```
-jprog
-board xupv2p
-target ppc_hw
```

```
-hw implementation/system.bit
-elf MontaVista_ELF/zImage.elf
-ace MontaVista_ACE/system.ace
```

Now, copy "<EDK_install_dir>/data/xmd/genace.tcl" (the default script used by Xilinx) to 'montaVista_GenACE/xupGenACE.tcl' and add the following lines to the most obvious place in the file:

```
# Production XUPV2P
set xupv2p(jtag_chain_options) "-configdevice devicenr 1 idcode
0x0127e093 irlength 14 partname xc2vp30 -debugdevice devicenr 1 cpunr 1"
set xupv2p(jtag_fpga_position) 1
set xupv2p(jtag_devices) "xc2vp30"
```

Finally, copy 'arch/ppc/boot/images/zImage.elf' to the 'montaVista_ELF' directory.

We are now ready to create the ACE file and store it to the CF card. If running the EDK under Microsoft® Windows, launch the Xilinx Xygwin shell either from the Start Menu or by running '<EDK_install_dir>/bin/nt/xbash.exe'.

Change directories within Xygwin to the Xilinx project directory. For example:

```
$ cd ../../../d/XUPV2P/linux_port/
```

Now, run the following command to create the ACE file[13]:

```
$ xmd -tcl montaVista_GenACE/xupGenACE.tcl -opt
montaVista_GenACE/xupGenACE.opt
```

The ACE file now needs to be stored on a DOS partition of the CF card. Make the following partitions on the CF card using 'fdisk'. Use 'fdisk -l' to determine to which '/dev' device the USB card reader was assigned. If the USB card reader is not shown then check to make sure that "USB Mass Storage" and "SCSI disk" support are compiled into the kernel and/or the proper modules are loaded.

- Partition 1: Type = FAT16 (6), Size = 32MB
- Partition 2: Type = Linux Swap (82), Size = 256MB[14]
- Partition 3: Type = Linux (83), Size = *Remainder*

For more information regarding the creation of the ACE file please refer to the SystemACE documentation [35]. Now that the CF card has been appropriately partitioned for the SystemACE controller and the Linux RFS we need to format them.

Command to format partition 1 on Linux (device may differ):

[13] For more information on exactly what the 'xmd' command is doing please refer to *Lessons Learned* section relating to the XST Flow.
[14] For most OSs the swap space should be approximately 1.0x – 1.5x the size of available main memory.

```
$ sudo mkdosfs -v -F 16 -R 1 -s 16 -n XLNX_ACE /dev/sdb1
```

Command to format partition 1 on Windows (drive letter may differ):

```
C:\> mkdosfs -v -F 16 -R 1 -s 16 -n XLNX_ACE F:
```

The above will **v**erbosely create a **F**AT**16** format file system on partition **1** of the CF card with **1** **R**eserved sector (the boot sector) and **16** **s**ectors-per-cluster, and **n**ame it **XLNX_ACE**. The SystemACE controller is very particular about the format of the ACE file data and requires that the partition be in the above format. The first partition will only be used to store the ACE file. Be sure to refer to "*Chapter 2: Using SystemACE for Non-Volatile Storage*" of the XUPV2P Hardware Reference manual [25] for commands specific to the size CF card.

Commands to format remaining partitions on Linux (device may differ):

```
$ sudo mkswap /dev/sdb2
$ sudo mke2fs -L XLNX_RFS /dev/sdb3
```

Copy the '`system.ace`' file created earlier to the FAT16 partition of the CF card, plug it into the CF slot on the board, ensure the SDRAM module is in place, launch the terminal software, and power on the board.[15] Something very similar to the boot log shown in *Appendix* section A.2 should be displayed. Please refer to the *Lessons Learned* section on the Linux boot process for more information regarding the output.

6.7 Building the RFS

Now that we have a minimal kernel up and running it might be nice if we could login to a shell, run a few simple informative commands, and take advantage of some other perks normally associated with every Linux distribution. To accomplish these minor tasks we will be using BusyBox to setup a simple RFS which the kernel will mount once it has loaded itself into main memory.

6.7.1 Setting up BusyBox

Download BusyBox [37], extract it to <`/home/username`> directory, and run:

```
$ make menuconfig
```

Make the minimum configuration changes to BusyBox as described in *Appendix* section A.3.

We now need to generate the directory and file structure of the RFS. To automate this process I have rewritten an older script, originally written by Wolfgang Klingauf [38], to work with the XUPV2P board. The '`mkrootfs.sh`' script I have created performs the following operations:

1) Removes any existing root file system in the location defined by `$RFS`

[15] It is not necessary to set the DIP switch (labeled SW9 on the PCB) to any particular setting as the flash card always has top priority in the SystemACE FPGA configuration device chain. If a valid ACE file exists on the CF card it will be used regardless of **any** other available configuration sources.

2) Creates all directories needed for a usable file system
3) Copies PPC cross-compiled GLibc to RFS
4) Copies over the '`/etc`' configuration files
5) Copies over a '`/www`' directory if it exists
6) Compiles BusyBox and stores it in RFS
7) Modifies permissions and ownerships of `$RFS`
8) Optionally creates a tarball of $RFS and places in `$RFS/..`
9) Optionally remakes the CF card's RFS partition and copies over the new RFS

Download and extract the '`mkrootfs.tar.gz`' tarball from my website. Executing the following command will display the script's help dialogue:

```
$ sh mkrootfs.sh -h
```

I suggest scanning through the script file in order to understand exactly what it's doing. As with Crosstool, do not run this script as '`root`' user!

Depending on the baud rate chosen for the UART (9600 baud is the only choice if using the Xilinx UARTlite), the following line in "`${RFS}/etc/inittab`" may require a slight modification:

```
::respawn:/sbin/getty 9600 tts/0
```

The `respawn` inittab "action" keyword tells the kernel to restart '`getty`' if the process is killed (i.e. we exit the shell). If we did not use this action we would never be able to login again once we have logged out. The '`getty`' is the application that opens a teletype port ('`tty`' or in this case '`tts/0`'), accepts the username, and then runs '`/bin/login`'. The '`login`' process then asks for the user's password. '`Getty`' and '`login`' could be the same application but they are separate for historical reasons. Upon login, the default shell which was previously defined in the BusyBox configuration will be launched.

We also need to modify one more file located in the kernel with respect to the UART16550. Change the following line in '`arch/ppc/boot/common/ns16550.c`' so that it matches the chosen baud rate:

```
#define SERIAL_BAUD     9600
```

6.7.2 Reconfiguring the Kernel

Having a root file system is nice but if the kernel has no idea how to mount it then it doesn't really do much good. Remember, we have merely told the kernel how it can communicate with the hardware in order to load itself. We will need to compile more drivers into the kernel before it is capable of mounting any type of file system. In the kernel configuration menu make the following change:

```
Block devices --->
      [*] Xilinx on-chip System ACE
```

This modification is the most important as it allows the kernel to treat the Xilinx SystemACE flash card as a block device (i.e. a storage device that can be read from and written to). This change will cause the following output to be added to the boot log:

```
Partition check:
 xsysacea: xsysacea1 xsysacea2 xsysacea3
System ACE at 0x41800000 mapped to 0xD1000000, irq=30, 1000944KB
```

Support is also required for the EXT2 file system format on which the RFS is stored. To do this make the following change:

```
File systems --->
      [*] Second extended fs support
```

If the kernel is loaded with this addition (and the last) it will cause the following output to be added to the boot log:

```
VFS: Cannot open root device "xsysace/disc0/part3" or 00:00
Please append a correct "root=" boot option
```

The kernel is now capable of mounting the RFS, but it still lacks complete support. Because the virtual terminal consoles are part of the '`/dev`' (device) file system we must also add support for it as well. This file system driver is still experimental so the "maturity level" option must be enabled in order to see it:

```
Code maturity level options --->
      [*] Prompt for development and/or incomplete code/drivers
File systems --->
      [*] /dev file system support (EXPERIMENTAL)
      [*]   Automatically mount at boot
```

If we do not include the option to **automatically** mount the '`/dev`' file system, the init process will not be able to open a terminal since '`console`' is listed in '`/dev`' as a device. In the more common Linux systems it is highly likely that the kernel will be configured with Framebuffer support (i.e. VGA/AGP), in which case it does not matter whether or not '`/dev`' is **auto-**mounted prior to the execution of '`getty`' since the console would attach to the D-SUB. To satisfy any curiosity, if the kernel is launched without '`/dev`' mounted the following will occur:

```
Warning: unable to open an initial console.
Kernel panic: Attempted to kill init!
 <0>Rebooting in 180 seconds..
```

Unfortunately, the '`/dev`' directory cannot be mounted manually within the '`/etc/init.d/rcS`' startup script because the '`init`' process attempts to display the following message to the screen when it is first executed:

```
init started:  BusyBox v1.2.0 (2006.08.04-07:34+0000) multi-call binary
```

But the above message cannot be displayed until we have a '/dev/console' and the attempted print to STDOUT results in the following crash and ultimately halts the system:

```
Warning: unable to open an initial console
```

Additionally, if POSIX shared memory objects are intended to be used in this design, support for the '/dev/shm' file system **must** be included as GLibc 2.2 and above require it – this document configured Crosstool for GLibc **2.3.3**. The PSM implementation on Linux kernel 2.4.X uses the '/dev/shm' dedicated file system. The appropriate option can be found here:

```
File systems  --->
     [*] Virtual memory file system support (former shm fs)
```

We must also ensure the '/dev/shm' directory is mounted as a temporary file system in '/etc/fstab'. The file in the '/etc' directory available on my website [2] is already configured for this:

```
tmpfs      /dev/shm        tmpfs   nodev,nosuid,noexec    0 0
```

The '/proc' (processes) folder is also critical for boot and **must** be included in the '/etc/fstab' file. We also need proc so that BusyBox can call itself through the '/proc/self/exe' symbolic link. Make this change here:

```
File systems --->
     [*] /proc file system support
```

And it too must be mounted via '/etc/fstab':

```
proc       /proc           proc    defaults               0 0
```

The '/proc' folder also works with the "Sysctl support" option so that making real-time changes to the kernel does not require a system call. I have included a common and very simple example below which disables all echo (ping) replies to echo requests:

```
$ echo 1 > /proc/sys/net/ipv4/icmp_echo_ignore_all
```

With that aside we are now ready to tell the kernel which partition to mount as the RFS. To do this make the following kernel configuration change (in bold):

```
General Setup --->
     [*] Default bootloader kernel arguments
          "console=ttyS0,9600 root=/dev/xsysace/disc0/part3 rw"
```

We also want to add the following to the '/etc/fstab' file to let the kernel know where the RFS will be mounted:

```
/dev/root     /     auto    defaults,errors=remount-ro      0 0
```

Basic support for process execution and dynamic kernel control is also displayed in the "`General Setup`" menu. These three options are not required for the RFS but are still highly useful. Specifically, IPC is used to allow processes to exchange information via shared memory. The `HELP` dialogue for these options is also quite self-explanatory. The interested reader can find them here:

```
General Setup --->
      [*] Networking support
      [*] Sysctl support
      [*] System V IPC
```

The resulting configuration for the kernel can be found in *Appendix* section A.4. It is now time to recompile the new kernel and create the new '`system.ace`' file using the skills learned in the previous sections.[16]

6.8 Testing the Complete System

We are now ready to test a "usable" embedded Linux OS. Copy the new '`system.ace`' file to the first (DOS) partition of the CF card. Also copy the entire RFS (the folders **within** `$RFS`) to the third (EXT2) partition of the CF card. Place the CF card in the appropriate slot, boot the board (ensuring that the SDRAM module is in place), and something very similar to the boot log shown in *Appendix* section A.5 should be displayed.

Congratulations! MontaVista's PowerPC Embedded Linux (complete with root file system) has just been successfully ported to the Xilinx Virtex-II Pro development board.

We can now retrieve some useful information from the kernel through the /proc file system. I will leave it to the interested reader to poke around but here is a simple example:

```
# cat /proc/cpuinfo
processor       : 0
cpu             : Virtex-II Pro
clock           : 300MHz
revision        : 8.160 (pvr 2001 08a0)
bogomips        : 299.82
machine         : Xilinx ML300
plb bus clock   : 100MHz
```

The minimum kernel configuration can be found on my website [2]. It will be called '`.config.MIN.RFS`'.

6.9 Adding Networking Support

[16] Remember to run any necessary shell export path commands for Crosstool before compilation if the shell has been logged out of recently.

First, add the following capabilities to the MontaVista kernel:

```
Networking options  --->
      [*] Packet socket
      [*] Unix domain sockets
      [*] TCP/IP networking
      [*]   IP: kernel level autoconfiguration
      [*]     IP: DHCP support
Network device support  --->
      [*] Network device support
Ethernet (10 or 100Mbit)  --->
      [*] Ethernet (10 or 100Mbit)
      [*]   Xilinx on-chip ethernet
```

Second, we will want to add some userland applications to determine the status of the Ethernet interface, check the kernel's routing tables, and fetch a DHCP IP address.[17] Add the following capabilities to the BusyBox RFS:

```
Networking Utilities  --->
      [*] ifconfig
      [*]   Enable status reporting output
      [*] inetd
      [*] netstat
      [*] ping
      [*] route
      udhcp Server/Client  --->
            [*] udhcp Client (udhcpc)
```

It's also a good idea to set the loopback and start the '`inetd`' daemon at startup. Add the following lines to the '`/etc/init.d/rcS`' startup script:

```
# Bring up loopback
$ECHO -n "Brining up loopback interface..."
/sbin/ifconfig lo 127.0.0.1 up
$ECHO "Done!"

# Start udhcpd
$ECHO -n "Starting udhcpc..."
/sbin/udhcpc --script=/etc/udhcpc/default.script
$ECHO "Done!"

# Start inetd
$ECHO -n "Starting inetd..."
/usr/sbin/inetd
$ECHO "Done!"
```

[17] Setting the interface address to a static IP does not require support for the micro-DHCP client. The IP address can be manual set by using '`ifconfig`'.

For the micro-DHCP client we need to create some simple scripts. These scripts can be found in the '`/etc`' directory available on my website [2] or simply retrieve them from the latest '`udhcpc`' tarball at site [39].

By default, Micro-DHCP will fill in '`/etc/resolv.conf`' automatically for DNS lookups, however, that default can always be changed in the script file. The DHCP client will also configure the interface and default routes as appropriate. If the DHCP client is not installed then these items must be setup manually. To perform a manual configuration, replace the DHCP client startup in the '`rcS`' file with the following:

```
# Set static IP address
$ECHO -n "Setting static IP address..."
/sbin/ifconfig eth0 up address 10.0.9.18 broadcast 10.0.9.255 netmask
255.255.255.0
/sbin/route add default gw 10.0.9.1 dev eth0
$ECHO "Done!"
```

If a unique (routable) MAC address is available then I suggest using it. A currently unused network card would suffice if only for temporary testing purposes. In fact, one might even try searching Google for "Ethernet thin net" and see if there is an old MAC address that is likely no longer in use.

The hardware MAC address of the Xilinx Ethernet_MAC can be changed by modifying the following lines in the kernel source code file "`arch/ppc/boot/simple/embed_config.c`":

```
bd->bi_enetaddr[0] = 0x00;
bd->bi_enetaddr[1] = 0x0A;
bd->bi_enetaddr[2] = 0x35;
bd->bi_enetaddr[3] = 0x00;
bd->bi_enetaddr[4] = 0x22;
bd->bi_enetaddr[5] = 0x00;
```

Change each of the 6 hex bytes to anything except for all `0`'s or all `F`'s. The default values represent the Xilinx preferred MAC address. If the reader does not plan on routing over the WWW (e.g. just a direct connection via a cross-over cable to another machine) then just leave the Xilinx default. If connecting two Xilinx boards to each other over the Ethernet port, be sure to make each MAC address different since all Xilinx boards will have the same default values.[18]

Boot the board and connect it to a viable network. If a routable MAC address is present and a ping to the internet is possible (e.g. '`ping google.com`') the board is in good shape.[19] One other thing that might be of interest is to configure BusyBox with '`rdate`' and sync the system clock to an '`rdate`' compatible server (e.g. '`rdate -s time.mit.edu`')

6.10 Setting up an HTTP Server

[18] A crossover cable will also be required, rather than a straight-through.
[19] This assumes there are no other firewalls on the user's internal network.

Configure BusyBox with the httpd daemon:

```
Networking Utilities  --->
      [*] httpd
```

Add the following to the '`/etc/inetd.conf`' file (my '`mkrootfs.sh`' will do this automatically):

```
www     stream  tcp     nowait  root    /usr/sbin/httpd httpd -h /www
```

The '`/www`' is the location of the website. This is similar to the '`/htdocs`' on other Linux distributions and it doesn't matter what the directory is called. Place a simple index.html file in the '`/www`' directory and, with the board successfully connected to a network, visit the board's IP address in any HTTP compatible browser.

6.11 Compiling Locally on the XUPV2P (Future Work)

I became quite tired of transferring cross-compiled binaries from my Linux box to the CF every time I wanted to install a new application. This will require bootstrapping the GCC with binutils on the Virtex-II Pro itself. The latest version of GCC that still includes the "*bootstrapping*" option is **2.9.5**.

Here is a rough outline of the procedure:

1) Configure BusyBox with:

```
Coreutils  --->
      [*] install
```

2) Cross-compile GNUmake and Binutils on x86 platform with Crosstool-chain and copy over to RFS – my 'mkrootfs.sh' script will do this but just be sure to change the GNUMAKE_SRC and GNUMAKE_INS variables

3) Bootstrap the GCC source on the XUPV2P board and finish the compilation procedure with a minimal GCC.

6.12 Setting up an SSH Server (Future Work)

After realizing that Sandia does not allow telnet communications on our internal network I decided to setup an SSH server on the XUPV2P board instead. Dropbear is a very small SSH server that is perfect for our tiny development system. Download the latest Dropbear source from this reference [75].

Compilation of the SSH server on the XUPV2P board will require a native GCC compiler. Proper operation of the SSH server will require editing the '`/etc/inetd.conf`' file.

6.13 Booting Linux via NFS (Future Work)

This will require an NFS server and NFS support to be configured in the kernel.

6.14 Booting Linux Kernel with PPCBoot (Future Work)

PPCBoot™ is just like the very common LILO and GRUB bootloaders but it is specifically designed for PPC embedded systems. PPCBoot™ would replace the ELF kernel image on the CF card and be loaded into SDRAM by the SystemACE controller. The bootloader would then load the ELF kernel image into SDRAM from a separate partition on the CF card. I will finish this in the near future.

6.15 Adding Support for IPv6 (Future Work)

This will be the perfect lead-in to my thesis as I will be working exclusively with IPv6 in the future.

6.16 Running a Simple IPv6 Chat Client (Future Work)

There is a very simple, open source IPv6 chat client which we can use to test our new kernel configuration.

7 Trials and Tribulations

This section covers the majority of the issues that I came across during this project. They are included in this document to (hopefully) save those working on the same project from unnecessary grief and frustration. They are listed in the order in which they were encountered.

7.1 "`UNDEFINED REFERENCE`"

This error was encountered when attempting to add networking support to the board. I received numerous "undefined reference" errors when trying to recompile the kernel with absolute minimal support for the Xilinx Ethernet_MAC IP core.

After quarreling with various driver combinations available for both MontaVista and Xilinx, I tried running '`make -d`' and was finally able to trace the errors back to the MontaVista `xilinx_syms.c` source file. As it turned out the file was including an older version of the Xilinx packet FIFO driver and not the one define by the BSP. After making the necessary changes described in the *Tutorial* section the kernel compiled perfectly.

7.2 Infinite Loop

This one is a classic case of not reading the datasheets. While I was initially learning how to use the Xilinx tools and becoming familiar with the board, none of the test applications I had compiled for the PowerPC would run. I first verified that the "Golden Configuration" worked properly and I then turned back to the Xilinx tutorials.

As it turns out there is one key setting within the XPS application development tab regarding what is called the "boot loop". The Xilinx boot loop application consists of a single "branch-to-self" assembly instruction that is meant to be enabled **only** when one of the following conditions is true:

1) The user would like to debug the board and ensure the PPC is calling the reset vector.
2) No user-defined application (that the user wants to execute at power-on) exists within BRAM.
3) The user intends to load an application into the off-chip SDRAM module through the JTAG port after the FPGA bit stream has been uploaded.

Although I had initially neglected to see the warning in the XPS Console Log, Xilinx does mention that a "memory overlap" exists between two enabled (i.e. "Mark to Initialize BRAMs" is set) applications within the "Applications" tab. The memory overlap occurs because no two applications can ever co-exist at the CPU's reset vector (e.g. for PowerPC that vector is address `0xFFFFFFFC`).

Instead of halting the build process, however, XPS continues and gives the boot loop priority to the reset vector over the user's application causing the user-defined code to never execute. In my opinion, this warning message should really be an error as this was quite likely never the user's intent.

7.3 Dual-Core Processing and Shared Memory

When I was first learning to write dual-processor applications for the V2P I had created a semi-tight-loop (i.e. a single second wait between each cycle) in which the first processor would check if a counter (written to by the second processor) had reached zero. Once the counter hit zero a message would be written to STDOUT.

It appeared that my application would write one instance of the value to the LEDs and then freeze (i.e. no change to the switches would be displayed). I then proceeded to single-step through the assembly code of the loop in real-time with the Xilinx modified GDB tool. I immediately noticed that the compiler had generated two very strange instructions for the loop. I have included two similar instructions here[20]:

```
      li    r5, *ptr
loop  bnz   loop
```

One would think that the compiler would generate the following:

```
loop  li    r5, *ptr
      bnz   loop
```

[20] None of the instructions in this section are proper PPC assembly instructions. They are meant only to demonstrate the issue at hand. The actual assembly code would not be nearly as meaningful.

Clearly, the compiler decided to completely disregard my loop (a.k.a. the compiler writer telling the programmer they are inferior ☺) and only ever check the counter once.

After finding more examples of dual-core applications via Google I noticed the frequent use of the word '`volatile`' when declaring variables. I had never even heard of this keyword before let alone used it in any application, hence, more research ensued and I have come to the following conclusion.

The '`volatile`' keyword is a qualifier that informs the compiler that the value of a variable may change at any time without any intervention by code the compiler finds [logically] nearby [40]. In this case, the counter variable is being modified by the second processor which is executing code in logically separate BRAM.

Thus, the compiler's choice to optimize out the loop was correct, but in doing so it broke the application because the '`volatile`' keyword was not used. The program executed as expected after including the keyword in the shared-memory pointer declarations. Disabling the GCC optimizations would have also solved the issue.

As a result of this not-so-nice experience I have turned off all GCC optimization options within Xilinx until I feel I can write "good" dual-processor, shared-memory code. It may be prudent of those just starting dual-core designs to do the same. The compiler flags can be modified in "**Software Platform Settings…**" under the "**Software**" menu.

7.4 "`Cannot open root device`…"

I received the following error during my initial attempts in trying to mount the root file system partition from the CF card:

```
VFS: Cannot open root device "xsysace/disc0/part3" or 00:00
Please append a correct "root=" boot option
Kernel panic: VFS: Unable to mount root fs on 00:00
<0>Rebooting in 180 seconds..
```

The kernel was stating that it was unable to mount the RFS. The "`root=`" option was already defined in the kernel's command string so I did not believe that to be the issue. After many tedious kernel configuration changes the kernel was finally able to mount the RFS. The key ingredient to curing this bug was adding support for the '`/dev`' file system. There are many reasons for needing the '`/dev`' file system but in this case it was required because the kernel links '`/dev/root`' to the specified partition (i.e. with '`root=…`') on the CF card.

Adding support for '`/dev`' also fixed the following message:

```
can't access tty: job control turned off
```

This error message was received when the user shell was initially launched by the '`getty`' process. This was a side-effect of lacking support for the '`/dev`' file system since the shell

could not attach itself to the virtual terminal in '`/dev/tts`'. If there is no terminal with which to accept user input then there is also no point in enabling job control.

The "job control" to which the message is referring is the ability for the user to send 'jobs' to the "background" so the user may execute other commands in the interim. Every command the user runs is a *job* that is executed. "Job control" consists of the following well-known shell commands: `fg`, `bg`, `kill`, `&`, `Ctrl-z`, `Ctrl-c`. For more information please refer to the following site [41].

7.5 Split Memory vs. Linux

This was one of the worst bugs that I ran into during this project and it occurred on an earlier version of the EDK software (i.e. service pack 2). When I first created the BSP for the MontaVista kernel build I configured the software platform settings to use all of the available 512MB of dual rank SDRAM. Under normal circumstances this would be fine, however, when XPS automatically generated the address ranges for main memory it split each 256MB rank into two non-contiguous address ranges.

In my initial attempt to boot the kernel I would only receive the four initial lines of address load information and then the system would freeze. After checking and re-checking documentation and various kernel options I finally turned to Google. I came across a forum on the PenguinPPC [42] site with another user that was having the same issue.

It turned out that the crash was occurring because Linux expects all of main memory to be within one contiguous address range – I had not initially noticed the non-contiguous addresses assigned by Xilinx. This requirement makes sense to me but it certainly is not a very well-known one to say the least. Needless to say, I manually changed the main memory address decode to fit each rank of memory within two adjacent address ranges and the kernel booted properly.[21]

7.6 The DOS/UNIX EOL Plague

I downloaded the widely used BusyBox to create a nice, simple, small RFS for my embedded Linux kernel. It compiled without any problems and generated a perfect directory structure. The only issue is that the '`/etc`' configuration files specific to the system must be created. I created the minimum required '`/etc`' files and thought that I was finished. This is where the fun began.

I looked on the web for anyone that had already created '`/etc`' files specific to an embedded system in order to give me some direction on what minimum files were required. I ran across the page of Wolfgang Klingauf [38] which had just what I needed. In trying to save some time and, yes, be a little lazy, I downloaded Klingauf's '`/etc`' directory and plugged it into my BusyBox RFS. I thought I could just modify these files to match my hardware setup and be on my merry way. This was a very incorrect assumption.

[21] After migrating from EDK SP2 to EDK SP3 I never saw the problem again. However, I have not exhausted all other possibilities for the strange address assignment. It might also have something to do with the enablement of a certain combination of peripherals.

I booted the board and when the ‘init’ process started it ran the startup script that I wrote from scratch perfectly, but as soon as it attempted to run ‘getty’ it started printing out partial log messages and all sorts of garbage. Regardless, it still gave me a login prompt so I tried to login as root (no password required as yet). I immediately received the 'motd' which I know is only displayed upon a successful login so I was quite excited. But that enjoyment was very short lived as I subsequently received some spectacular "kernel panic" messages:

```
init started:  BusyBox v1.2.0 (2006.07.31-02:46+0000) multi-call binary
'tarting pid 10, console /dev/console: '/etc/init.d/rcS
': No such file or directoryinit.d/rcS
': No such file or directoryinit.d/rcS
Starting pid 11, console /dev/console: '/sbin/getty'
: No such file or directory
' (pid 11) exited.  Scheduling it for restart.
Starting pid 12, console /dev/console: '/sbin/getty'

------------------------------------------------------------

: No such file or directory
(none) login: root

Welcome to the XUPV2P, EIS edition

login[29]: root login  on `console'

: No such file or directory
Kernel panic: Attempted to kill init!

(none) login:
```

Notice the ": No such file..." error message with absolutely no information on what exact directory or file it thinks is missing. No, that is not a copy and paste error, the ‘init’ process gave me nothing to work with! It then correctly proceeded to respawn the ‘getty’ process and gave me another login prompt. I attempted to login two more times and the exact same kernel panic occurred every time.

To troubleshoot the issue I decided to try two different earlier versions of BusyBox (as the one I am currently using is a stable/development version) with various configuration options. I also thought to try manually adding a user to the ‘/etc/passwd’ file and logging in as someone other than root, but it was to no avail. This lead me to believe it was a kernel configuration issue rather than software. Hence, I also tried a few different kernel configurations and the exact same thing would happen.

After doing some fruitless research online I stumbled across a toolset called TinyLogin which is used quite often as a nice complement to the BusyBox RFS setup. It has some more bells and whistles when it comes to user and information security (e.g. ‘/etc/shadow’). Most importantly, it provides its own ‘getty’ and ‘login’ processes so I thought it would be a good way to rule out some of the software bugs I was contemplating. However, TinyLogin crashed and respawned just like BusyBox.

I also thought that it might be a permissions issue in the root file system so I tried different combinations of the following commands but none of them made any difference.

```
chmod -R 4755 /
chmod -R 4777 /
chown -R root:root /
chown -R root:users /
```

The last thing I could think of was with the last major problem I had solved a couple days ago where the login prompt wouldn't even appear at all. It turned out that the '`/etc/inittab`' file was in DOS format rather than UNIX. Apparently Klingauf (from whom I retrieved the '`/etc`' directory) was developing on a DOS based machine. I thought for sure that couldn't be the problem since I had already converted every file in the '`/etc`' directory. I opened each file in the '`vi`' editor on my Gentoo box and I saw no infamous '`^M`' characters representative of a DOS format file.

After three days of complete frustration, little sleep, and finally giving up, I decided to move on to other things. I figured I could still ask a co-worker about it (a different pair of eyes always helps) when I went back to the labs. I knew that I could still pass the '`init=/bin/sh`' command to the kernel during boot and at least get a shell where I could poke around and work on other things. The bad thing about that is that my '`/etc/inittab`' file is not parsed and, consequently, if I exit the shell '`getty`' will not be respawned - a one shot deal! There was also no way for me to login as any specific user (i.e. I'm stuck being root) since the '`login`' process also fails with a kernel panic.

I finally booted the board with the new kernel command and received a beautiful shell prompt. I went to edit the Bourne shell '`/etc/profile`' file to make some minor changes, AND there were '`^M`' on every line - **it was still in DOS format!** I was in total disbelief. I checked the original file on my Gentoo box and there were no '`^M`' characters. I thought that my script had not been copying over the files properly but the modification timestamps on the CF card were correct.

To make an already long story short, it turned out that running the '`dos2unix`' to convert all of the files in the '`/etc`' directory had failed. As soon as it reached the '`/etc/init.d`' directory the process terminated because it was not a file. All of the files that came alphabetically after '`init.d`' in the '`/etc`' directory were never converted. I missed the error message amongst the hundreds of other compiler statements and various other script commands that occur when '`mkrootfs.sh`' is run.

I converted the remaining files and booted the board and the problem was gone. Apparently the '`init`' process did not appreciate trying to execute <insert_process_name_here>CR. Where 'CR' is "carriage return" appended to every process name – which also explains the incomplete error messages. That is, instead of displaying the missing process (e.g. '`/sbin/getty`') it wrote the line to the terminal with an appended 'CR'. Thus, moving the cursor back to the beginning of the line and finishing with ": No such file or directory". A wonderful effect, no??

What I still didn't understand is why '`vi`' wasn't displaying the '`^M`' characters in the file. After doing some research online it turned out that my version automatically detects if the file is in MS-DOS format and ignores the 'CR' (i.e. does NOT display them). The only way to have them displayed is by using the following special command which overrides the displayed file format:

```
:edit ++ff=unix file.txt
```

Or the above change can be made permanent in the '`/etc/vim/vimrc`' file:

```
set fileformats=unix
```

The three file format names that Vim uses are:

```
unix  LF
dos   CR LF
mac   CR
```

More information on the '`vi`' file formats can be found in the user's manual [43].

8 Lessons Learned

Many new concepts are being learned while working with on this project. These concepts range from various hardware implementation methods to the inner workings of the Linux operating system. They are included in this document to help the reader become more familiar with some advanced aspects of hardware and software design.

8.1 ECC Memory

ECC memory utilizes special circuitry that verifies the accuracy of the data passing in and out of the memory chips on the memory module. ECC is capable of detecting and *sometimes* correcting some data transfer errors. If a single-bit error lies within a 64-bit memory block then ECC can correct it, but anymore than that and ECC will only be able to detect that the error has occurred. ECC is used because all RAM will occasionally experience errors due to factors such as [44]:

- RF interference
- Power fluctuations
- Static discharges
- Faulty components
- Improper system timing

In order to use ECC all modules within the system must be ECC compatible (i.e. no mixing and matching), ECC must be supported by the motherboard, and ECC must be enabled in the BIOS. ECC memory is usually used in large servers (where accuracy and stability are key) rather than smaller home-use systems. This is mainly because it is slower and more expensive.

An ECC memory module is identifiable by simply counting the number of chips on the memory module. If the number of chips is odd then it is ECC memory, otherwise it is non-ECC. There will also be 72 pins (less pins for V_{DD} and V_{SS}) defined in the datasheet as opposed to 64. The extra (9^{th}) chip contains the single extra parity bit for every byte located within each of the other eight chips [45]. It is up to the SDRAM controller and OS to use those extra bits to its advantage.

It is a common misconception that a different chip located on the outer edge of the PCB, correctly known as the SPD-EEPROM, is an indication of ECC memory, but that is simply not the case. The first 128 bytes of the SPD-EEPROM (small 8-pin, 2Kb chip) contains information specific to the memory module such as timing parameters and memory organization. The remaining 128 bytes may be programmed by the user. The timing specifications for programming the chip are defined in the same datasheet as the module itself.

The SER of modern memory circuits is now low enough to place non-ECC memory as the norm. However, where extremely high-reliability is required for mission-critical data, even a 1% chance of failure for the lifetime of the module may be too great to allow for a non-ECC module. And, of course, in the context of this document, ECC memory is also a great learning tool for parity and error detection/correction.

8.2 Ranked Memory (and its effect on Linux)

From my research on the internet during this project I have come to find that there is much confusion to what exactly a memory "rank" refers. It is important to understand what ranked memory is as the XUPV2P board will only accept certain types and we don't want to waste time or money on incompatible memory. I will attempt to clarify some of these misunderstandings in this section.

Some have confused memory *ranks* with memory *banks* and vice versa. Memory rank has more to do with the physical construction of a memory module rather than the internal logical division (i.e. a memory bank). Specifically, ranked memory refers to the density of each individual DRAM chip on the memory module. The more ranks there are per memory module the lower the DRAM chip density. The number of ranks has nothing to do with the memory capacity (e.g. MB, GB) of the module. See Figure 15 for an illustration of the various available memory rankings. Assume that each of the ranks shown have the same memory capacity.

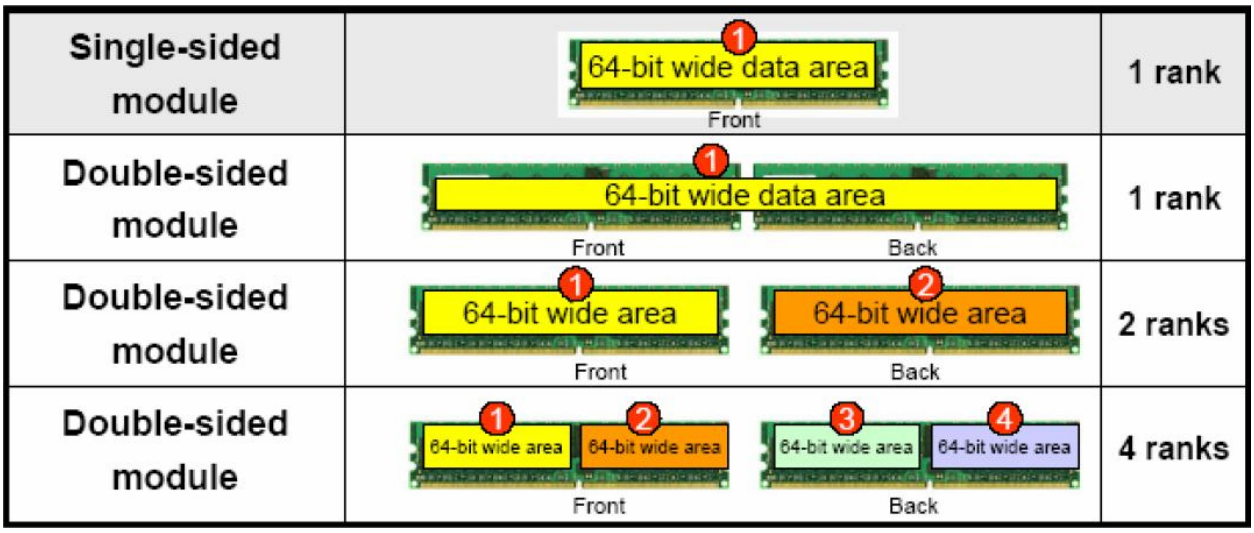


Figure 15: How ranks can be applied to a DIMM [46]

There are currently three different types of ranked memory: single, dual, and quad. Memory modules are manufactured with higher ranks to reduce the cost to the consumer. It's quite hard to receive good die yield for higher density chips, which is why they cost more. Most computer architecture course speak of die yield vs. cost. Die yield refers to the number of good die divided by the gross die on the original wafer. Thus, the higher the die yield the lower the cost.

This is true for any type of die whether it be for DRAM cells or CPUs. For example, on CPUs the manufacturer can decrease the cache size to reduce the cost of the chip. This is one of the main differences between an Intel Pentium processor and an Intel Celeron processor; the Pentium 4 may have up to four times the amount of available cache memory.

There is also a drawback to purchasing cheaper, high-rank memory as all motherboards have a maximum number of ranks they can handle. For instance, if the motherboard only supports a maximum of four ranks and one quad-ranked module is purchased, then it will not be possible to purchase more modules to upgrade the system's memory capacity. This is the case even if there are empty DIMM slots on the board. The XUPV2P board only supports single or dual ranked modules. Refer to the motherboard's instruction manual to determine the maximum number of ranks for that board.

Few motherboards even support quad-ranked memory. High-rank capabilities are more common to high-end servers which usually contain massive amounts of memory. By creating a motherboard that is able to control numerous ranks (e.g. 16) the board manufacturer decreases the memory upgrade cost to the consumer since they can buy cheaper, high-rank memory but with the same capacity as, say, a single-ranked module.

For a great explanation of DRAM operation please reference the following site [47].

8.3 DDR Memory and Dual-Pumped Architectures

DDR memory differs from SDR memory in that data is transferred on every clock edge. This is accomplished by giving the memory module an internal data base which is twice the width of the external data bus. For 32-bit bus architectures this means that the DIMM will have a 64-bit internal data bus. For every bus clock cycle 2 32-bit chunks of data will be read – this is referred to as a "2*n*-prefetch architecture".

A VHDL/Verilog programmer might ask themselves how it is possible to transfer on both the rising and falling edges of a clock in a proper synchronous design. Clearly, we cannot create a logic block that triggers on both the positive and negative edges of a single clock. To remedy this issue the DDR standard requires a pair of differential clock inputs. One clock is the true bus clock and the other is its complement (i.e. 180° out of phase). The first 32 bits of data in an x64 DIMM are transferred on the rising edge of the true bus clock while the remaining 32 bits are transferred on the rising edge of the inverse clock. Table 3 lists five DDR SDRAM modules with various speed ratings.

Type	Base FSB Frequency	Effective RAM Speed
PC1600 or DDR200	100 MHz	200 MHz
PC2100 or DDR266[22]	133 MHz	266 MHz
PC2400 or DDR300	150 MHz	300 MHz
PC2700 or DDR333	166 MHz	333 MHz
PC3200 or DDR400	200 MHz	400 MHz

Table 3: DDR Types

The ability to provide the same throughput at a slower clock speed has tremendous advantages. Theoretically, the bandwidth of a DDR memory module connected to a FSB running at 100MHz would be equivalent to that of an SDR memory module connected to a FSB running at 200Mhz. Unfortunately, this is not the case as we must also consider the protocol overhead associated with dual-pumped architectures. For example, the Intel® Pentium 4 requires an MCH to access main memory appropriately [48]. A small decrease in throughput, however, is a small price to pay for a reduced clock speed; slower clocks reduce heat, increase stability, and lower power requirements.

Although it is not available on the XUPV2P board, if more that one DIMM slot exists it is very important to put the largest memory module in the lowest slot number. The reason behind this is that the operating system will fill memory beginning with the lowest slot and ending with the highest. If the smaller memory module were placed in slot 0 then the OS would have to access both memory modules sooner than necessary, thus, decreasing total throughput.

There is also QDR, XDR, and various other speed memories which will not be discussed here. However, the interested reader may find some useful information in [50]. In addition to increasing throughput via *n*-pumped architectures there is another alternative known as "self-clocking" which is used by PCI-E [51] and InfiniBand [52] technologies. Please refer to the following reference for more information [53].

[22] Supported by the XUPV2P [49].

For more information on DDR memory and its functionality please refer to the following functional specification provided by Micron [54].

8.4 Linux Boot Process

Although the Linux boot sequence is fairly straightforward, what happens behind the scenes from the moment the power button is pressed until the login prompt appears is taken for granted by most users. This is quite understandable considering that ignorance of the boot process does not prevent the user from actually *using* the operating system. However, for an open source software/hardware developer (and especially for this project) comprehension of the boot sequence is critical. I will explain the differences between booting the XUPV2P board and booting a normal desktop/laptop motherboard as they arise.

When the board is first powered on it needs to learn the location of the operating system. The first piece of hardware involved in this process is the BIOS chip. The BIOS chip accesses a section of the hard disk (or whatever location specified to check first in the BIOS setup) called the *boot sector*. Unfortunately, the boot sector is quite small and a modern kernel cannot fit inside it. But we are one step closer to finding the operating system because within the boot sector lays the *boot loader* (e.g. GRUB, LILO). The boot loader is executed, finds the true location of the kernel (i.e. not in the boot sector), and loads it into main memory.

On the XUPV2P board, however, there is obviously no BIOS chip…or is there? Well, not exactly, but the same basic concept still exists and the logic is located in the Xilinx SystemACE chip. The SystemACE chip is a highly intelligent device. It uploads the hardware BIT stream to the FPGA and then uploads the ELF kernel image to SDRAM. See the section entitled "Loading the Application into the Target FPGA" of [1] for more information on the SystemACE design flow. I have also included a much more detailed description of the file and source code conversion of this entire process which can be found in the *Lessons Learned* section for XST flow.

Besides allowing Xilinx SystemACE to directly upload the kernel to main memory (the recommended method) we can also take a more traditional approach by using the PPCBoot™ bootloader. Please see the *Tutorial* section for more details.

Once the kernel has made its way to main memory the boot process is like any other Linux system, embedded or otherwise, running over a serial port. The kernel first parses its command string and configures the serial port for the given speed and destination:

```
Linux/PPC load: console=ttyS0,9600 root=/dev/xsysace/disc0/part3 rw
```

The kernel will then go about loading all of the configured drivers and support with which it was originally compiled and initializing each device. Remembering that most operating systems never reference effective addresses, Linux will remap all of the I/O devices to Virtual Memory using a function called '`ioremap()`'. The result of this is shown in the kernel boot log on the following lines:

Interrupt Controller remapping:

```
Xilinx INTC #0 at 0x41200000 mapped to 0xFDFFE000
```

UART_16550A remapping:

```
ttyS00 at 0xfdfff003 (irq = 29) is a 16550A
```

Xilinx SystemACE remapping:

```
System ACE at 0x41800000 mapped to 0xD1000000, irq=30, 1000944KB
```

Xilinx Ethernet_MAC controller remapping:

```
eth0: Xilinx EMAC #0 at 0x40C00000 mapped to 0xD1013000, irq=31
```

The next step in the boot process involves mounting the RFS as specified in the kernel command string. Remember, once SystemACE finished loaded the kernel image from CF to main memory it forfeits control of the system and acts as any other block device:

```
Linux/PPC load: console=ttyS0,9600 root=/dev/xsysace/disc0/part3 rw
VFS: Mounted root (ext2 filesystem).
Mounted devfs on /dev
```

The final step for the kernel is to execute the first userland program. The first program run is usually '`/sbin/init`' since it is the default if no other application is referenced in the kernel command string with the '`init=`' option.[23]

The first task for '`init`' is to parse the '`/etc/inittab`' file and execute whichever application has been assigned to the "sysinit" *action*. The "sysinit" action command is usually a startup script. For example, the '`/etc/init.d/rcS`' startup script that I created for the XUPV2P board performs the following tasks:

1) Mounts the '`/proc`' file system
2) Re-mounts '`/`' as read-write
3) Mounts '`/dev/shm`' for POSIX shared memory
4) Starts `syslogd` & `klogd`
5) Sets the board's hostname
6) Sets up loopback (127.0.0.1)
7) Fetches a DHCP address for `eth0`
8) Starts `inetd`
9) Sets the system clock to current time via `rdate`
10) Starts web server daemon

Once the command or script assigned to "sysinit" is complete the '`init`' process will execute all applications assigned to run-level 1. There are usually only two programs assigned to run-level 1:

[23] If '`/sbin/init`' cannot be found after the RFS has been mounted and no other program has been specified, the kernel will crash right after it frees any unused kernel memory…sometimes without any error message.

1) ‘getty’ – used to bring up a login prompt
2) The program to be executed when “Ctrl-Alt-Delete” is pressed (e.g. “shutdown –r now”)

Steps 1-5 of the ‘rcS’ script can be considered run-level 1. Steps 6-10 would be considered run-level 2. Run-levels 1 and 2 are also known as “single-user-mode” where the only user that may log in is ‘root’, only one instance of ‘root’ is allowed. The only difference between levels 1 and 2 is whether networking support exists. Run-level 1 provides a basic maintenance shell that can be used for repairs. The system will always be in one of the seven states shown below:

0 = halt
1 = single-user
2 = single-user with networking support
3 = multi-user mode
4 = not define
5 = multi-user running Xwin
6 = reboot

Once the system has successfully booted to run-level 2 we can enter what is known as the “default run-level”. The default run-level is usually set to one of the multi-user run-levels (3-5) via the following command line in the ‘/etc/inittab’ file (where ‘X’ is a number 2-5):

```
id:X:initdefault:
```

The default state could be set to single-user mode (1) for system repair purposes; however, a multi-user run-level is the most common state for the system since it allows non-root user logins. The *normal* ‘init’ process will execute all commands assigned to the specified default run-level. However, BusyBox ‘init’ process does not understand run-levels.

BusyBox is meant to be small and as “bare-bones” as possible. Since there is really no need for multiple run-levels on our tiny embedded system this isn’t a problem. However, if run-level support is required, one can always download and compile a true ‘init’ application and add it to the ‘/sbin’ directory of the RFS. For more information specific to the XUPV2P ‘inittab’ file please refer to that file which is available on my website [2].

Once the ‘init’ process has successfully executed all of the commands, it will sit and wait in memory so that any applications which have been assigned to the *respawn* action command can be re-executed. In addition, ‘init’ will always have a PID of ‘1’ since it is the first program executed by the kernel. If a different application was assigned to the kernel command string with ‘init=’ then that program will have a PID of ‘1’ instead. It is highly recommended to use the default ‘init’ process.

Much of the above information was learned from [55] and [56]. For more information on the Linux boot process I suggest starting with them.

8.5 BSD vs. GNU Licensing

If a designer intends to create a proprietary design when using open source code it is important to know the major differences between BSD and GNU licensing. This is especially important in my case since I work for a government laboratory. Unfortunately, when I begin developing hardware and software for Sandia network security on the Force10 box (mentioned in the *Motivation* section) that source code will be proprietary and not available to the general public.

This is where the differences between BSD and GNU GPL licensing come into play. Even though both licenses consider their software to be "free" they are extremely different in the way original source code may be used. BSD licensed software releases may be built into any proprietary, closed-source code with the only major stipulation being that the original authors must be named. The GNU GPL on the other hand requires that any "derivative work" must also be released under the GNU license, thereby disallowing any closed-source material [57].

However, a common misunderstanding of the GPL license is that usage of the core functionality of the Linux kernel (i.e. system calls or dynamic linkage) is not considered "derivative work". The GNU C Library, which is released under the LGPL, provides an interface to the kernel and linking software though it is also free of any GPL licensing requirements [1].

8.6 PowerPC™ DMIPS Benchmarking (Future Work)

The PowerPC 405 does not include an FPU co-processor so Dhrystone benchmarking is the weapon of choice. I will also include a Whetstone benchmark using the SoftFloat and kernel FPU emulators.

8.7 The IBM CoreConnect™ Bus

The IBM CoreConnect™ Bus architecture is used to interconnect various peripherals to the IBM PowerPC processor and to each other. Any peripheral core wishing to connect to this Bus must implement the CoreConnect communication protocol standard. These devices may consist of DMACs, PCI controllers, interrupt controllers, etc. [58]. Each of the devices on PLB/OPB bus must be designated as either a master or a slave. Figure 16 provides an illustration of the PLB-OPB implementation of this architecture.

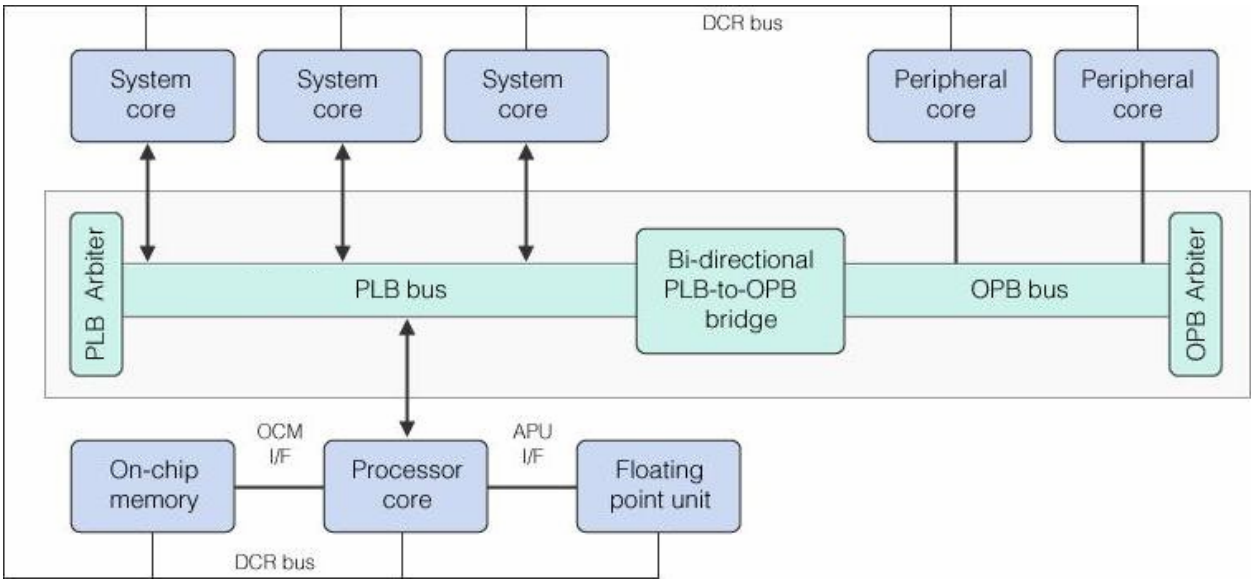


Figure 16: IBM CoreConnect™ Bus Architecture [59]

Xilinx supports the CoreConnect bus architecture on both the soft core MicroBlaze™ processor and the hard core PPC405 processor. Figure 16 shows that the CoreConnect architecture consists of the following six major components [60]:

- **PLB** – A fully synchronous bus meant for high-bandwidth, high-performance devices (e.g. DDR SDRAM). This bus is connected directly to the processor core and provides a 32-bit address bus and a 64-bit data bus.

- **OPB** – A fully synchronous bus meant for low-bandwidth, low-performance devices (e.g. UART). This bus sits at a separate level of the bus hierarchy and operates completely independently. The OPB provides 32-bit address and 32-bit data buses.

- **DCR** – This bus is used to transfer data between the registers within the various PLB/OPB masters/slaves. It can also be used to read/write data to/from the CPU's GPRs. The DCR's main purpose is to decouple the PLB from low-speed read/write transfers generated by the OPB.

- **PLB-to-OPB Bridge** – This logic block allows data transfer between the PLB and OPB devices through any PLB device designated as a master.

- **PLB Arbiter** – Logic block needed to configure the PLB and provides address/data flow control.

- **OPB Arbiter** – A byte-wide slave device used to configure the OPB.

The CoreConnect architecture results in all peripherals being mapped to the main 32-bit system memory address range of the PowerPC (see *Appendix* section A.8 for the memory map of our system). This is in contrast to x86 bus architecture in which there is physical "`M/IO`" output pin which signals whether a memory or I/O peripheral operation is occurring (see pin 28 of Figure

17). The x86 implementation also requires the use of special I/O instructions (e.g. `in`, `out`) and a port address register (i.e. DX).

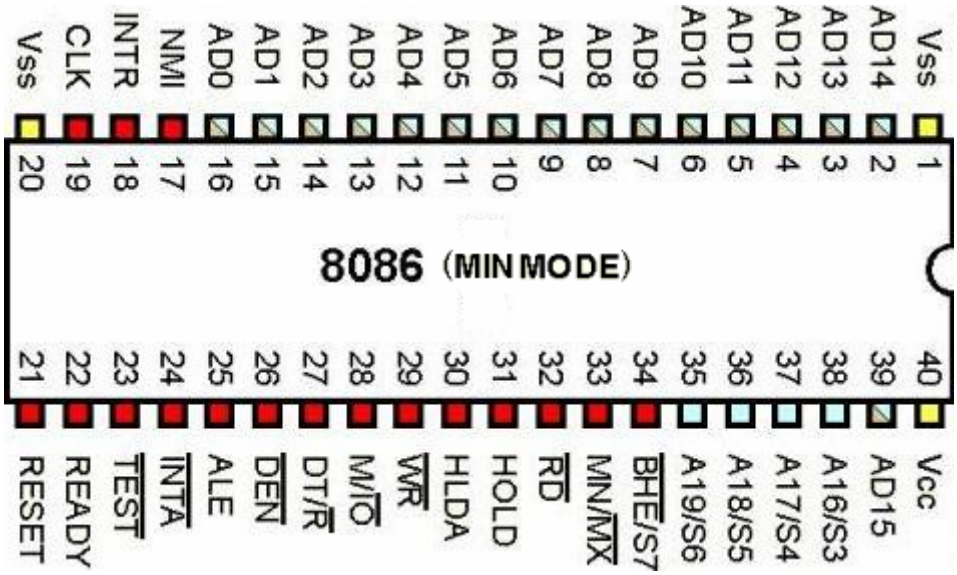


Figure 17: 8086 (Min Mode) Pin-out [61]

8.8 DLL vs. PLL (a.k.a Xilinx vs. Altera)

The world of digital clock management in embedded systems is a very competitive one. There will always be advantages and disadvantages to similar technologies with the same ultimate goal of removing clock delay. In this case it is always best to take a step back and determine which one is the best based on the application at hand. I include a few well-known facts on both technologies here.

Altera Corp. uses the PLL as their method of choice for DCM. PLL technology is capable of generating non-integer multiples of the input clock, but it requires an analog implementation in order to do so. Analog devices, however, consume less silicon real-estate to implement. "Phase error" can also accumulate during initial input thereby increasing lock acquisition delay; however, the phase capture range is unlimited.

Xilinx Inc. utilizes the DLL clock multiplication technique. The DLL is a fully digital implementation making it easier to work with in digital design, however, as a consequence, it only supports integer multiples of the input clock. The digital design does allow for low power consumption and low jitter. "Phase error" does not accumulate on initial input providing for a fixed lock acquisition timing parameter; however, DLLs have a limited phase capture range.

Regardless of the information I have presented here, I suggest the reader perform their own research before jumping to any conclusions. A few good starting points can be found in [62], [63], and [64].

8.9 The XST Flow

The work that Xilinx goes through to generate the bit stream for the design (hopefully within the timing constraints) that is ultimately mapped to the FPGA is quite complex – and usually taken for granted. After porting Linux to the XUPV2P board I have a much better understanding and appreciation for the process. I have included my understanding of the design flow here to help anyone reading this document better understand the Xilinx FPGA creation methodology.

The XST flow can be visualized in various ways but the following are the two that I like the most. The first is a more general portrayal created by Xilinx and the other is more file specific which I created myself.

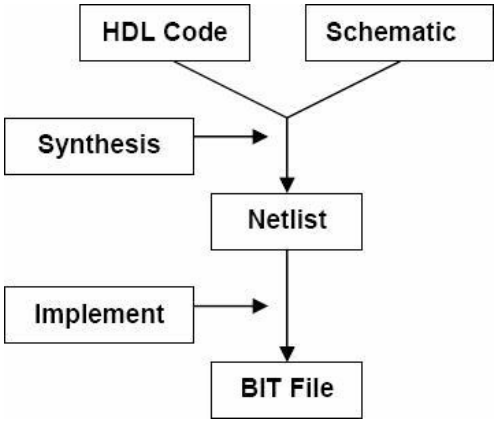


Figure 18: Xilinx Design Flow (1) [65]

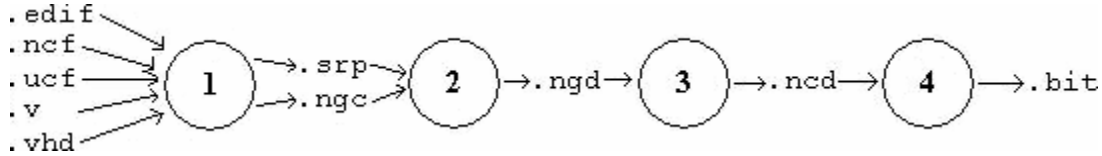


Figure 19: Xilinx Design Flow (2)

The diagram I have created in Figure 19 illustrates the Xilinx design flow in four steps:

1) Translation – Xilinx tools translate each of the user's input files into a single common file (NGC) and generates the synthesis reports for the user's review (SRP)

2) Reduction & Accommodation– design is reduced to Xilinx-only primitives (NGD) and accepts input from the schematic editor and other 3rd party synthesis tools.

3) Mapping – each primitive is mapped to specific device resources (NCD) (e.g. logic cells, I/O cells, slices)

4) Place & Route – resources in the NCD file are assigned to specific locations within the FPGA and the connections between each resource are mapped into the device's interconnect fabric. The delays associated with an FPGA's interconnect network can

be significant! Therefore, the PAR algorithms are the final determining factor in whether or not the user's design will meet the timing constraints given.[24]

Once the '`.bit`' has been generated we now have three methods of programming the FPGA: via PROM (5a), via USB/PC4 JTAG port (5b), via CompactFlash (5c).

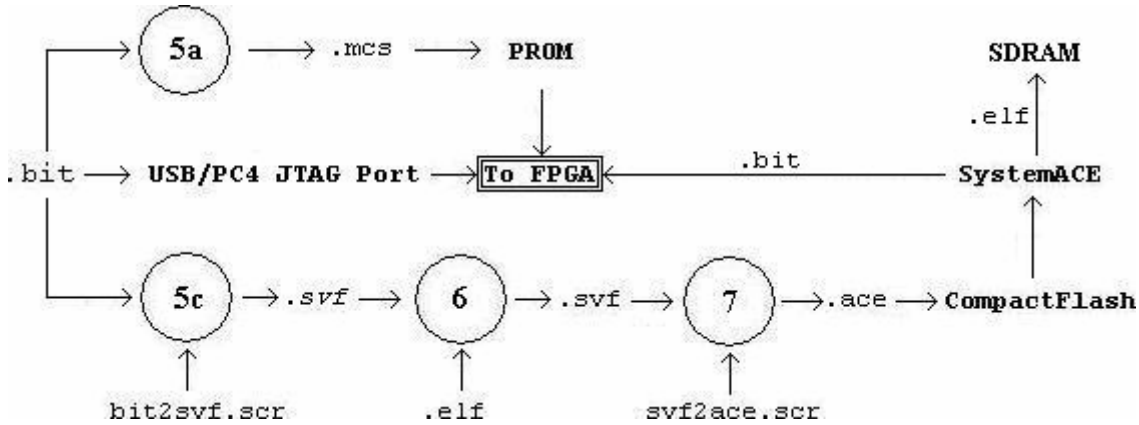


Figure 20: Xilinx Design Flow (3)

5) a) PROM – Xilinx PROMGen is executed which creates an Intel MCS-86 ('`.mcs`') file from the original '`.bit`' file which is written to the board's PROM chip. The PROM file can then be uploaded to the FPGA at power-on.

   b) USB/PC4 JTAG Port – This method will upload the raw '`.bit`' file directly to the FPGA. This method is best for very quick debug/test purposes since the '`.bit`' file was never stored anywhere permanently (e.g. PROM, CompactFlash) and the FPGA will only store that hardware revision until the board is powered off or reprogrammed.

   c) CompactFlash – Xilinx iMPACT converts the BIT file into an SVF file. The first SVF version, however, does not yet contain any ELF software binaries (in our case the kernel), it is merely a conversion of the original '`.bit`' file using the `bit2svf.scr` script file.[25]

6) XMD (1) – XMD now takes the ELF binary and adds it to the preliminary SVF file merging them into one common SVF.

7) XMD (2) – XMD now converts the final SVF file to SystemACE-compatible format using the '`svf2ace.scr`' script file.[26] The ACE file is then written to the CF card and read by the Xilinx SystemACE IC at power-on. Once the BIT file contents of the

[24] If the specified design constraints are too tight the PAR algorithm will give up, **but** it will still create a valid bit stream (just not one that meets constraints). Hence, the user should **always** check every PAR warning message.
[25] `bit2svf.scr` is automatically generated by XMD based on the '`.tcl`' and '`.opt`' files created in the Approach section.
[26] `svf2ace.scr` is automatically generated by XMD based on the '`.tcl`' and '`.opt`' files created in the Approach section.

ACE file have been uploaded to the FPGA, SystemACE uploads the ELF kernel image contents of the ACE file to SDRAM.

Please refer to the sections in the *Appendix* regarding Acronyms and File Descriptions.

8.10 Construction of an Ethernet Interface

When I was first learning to interface with the 10/100 Ethernet MAC hardware on the board I came to realize just how little I knew about the actual construction and terminology invested in a *simple* network port. I have included a small amount of what I have learned here so the interested reader can get a head start in understanding network connectivity with the XUPV2P board and with other network devices in general.

10/100 Ethernet is a network protocol defined by the IEEE 802.3 standard [66]. The Ethernet interface on the XUPV2P board and other networking devices consists of three main components: the MAC, the PHY and the Ethernet coupling magnetics.

On the XUPV2P board the MAC is the IP core which lies inside the FPGA. It acts as the interface between the LLC and physical layers of the OSI model. The MAC determines who is allowed to access the physical media at any one time [67].

The PHY, PHY-chip, or "PHYceiver" is the functional block that encodes/decodes data between digital and analog. This is the part of the interface that lies at the physical layer of the OSI model. It is usually physically coupled with the MII block in a single IC. On the XUPV2P board this is component U12 (Intel LXT972A)

There is another logic block, referred to as the Media Independent Interface (MII), which is used to interconnect the MAC with the PHY. "Media Independent" means that the same MAC block can be connected to several different types of PHY chips [68]. Equivalent terms for the MII at various speeds are: AUI for 10/100Mbps, GMII for 1Gbps, and XAUI for 10Gbps.

A standard MII features the following set of programmable registers [68]:

- Basic Mode Configuration
- Status Word
- PHY Identification
- Ability Advertisement
- Link Partner Ability
- Auto-Negotiation Expansion

The MII status word is the most useful as its value can determine whether or not a host is connected to the network. The register contains a bitmask with the following meanings [68]:

```
0x8000 Capable of 100baseT4
0x7800 Capable of 10/100 HD/FD (most common)
0x0040 Preamble suppression permitted
```

```
0x0020 Auto-negotiation complete
0x0010 Remote fault
0x0008 Capable of Auto-negotiation
0x0004 Link established
0x0002 Jabber detected
0x0001 Extended MII register exists
```
[27]

8.11 Serial Communication with the XUPV2P Board

I have included this section mainly for those that are developing solely with the Linux operating system. There are many methods for communicating with the XUPV2P board over the serial port on Linux (some of which may be a bit convoluted) while it is extremely simple to do so in Windows (e.g. Tera Term Pro or HyperTerminal).

A few GUI serial communication programs available for Linux include Minicom, Seyon, Kermit, etc.. If running Gentoo simply '`emerge`' any one of them. One could also write a very tiny Python script using the PySerial extension to link to the board. The simplest way that I have found, however, is to run the following command in a terminal:

```
$ cat < /dev/ttyS[01]
```

This will re-direct all input from serial port 0 or 1 to STDOUT and display it to the terminal. To send information out of the port (i.e. user input) simply run the same command with the re-direct pointed in the other direction in a second terminal window. The responses to the commands will appear in the first terminal window.

9 Future Work

The opportunities for hardware and software design with this board are limitless. I will definitely be adding more information related to networking support on the XUPV2P board. The MontaVista Linux 2.4 kernel has support for IPv6 and I intend to run a small IPv6 chat client between two XUPV2P boards.

I will also be adding a section on using the PPCBoot™ bootloader mentioned in the *Lessons Learned* section. This will provide for a more traditional bootstrapping method and understanding of the kernel loading and execution procedure. In addition, I will be adding a *Lessons Learned* section on understanding the Linux RFS [70] and a *Tutorial* section on the SoftFloat math co-processor emulator [71].

Relating back to the *Lessons Learned* section on BSD vs. GNU licensing I will also attempt to port a BSD kernel to the XUPV2P board as well. This will aid in my understanding of the major differences between the GNU and BSD kernels for my thesis research [18]. I will also be adding *Tutorial* and *Lessons Learned* sections on the MicroBlaze™ and PicoBlaze™ soft processor cores, including how to instantiate them and the differences between them.

[27] The Intel LXT972A PHY on the XUPV2P board does have extended register capability and is used to configure the interface for either CSMA/CD or full-duplex operation [69].

10 Conclusion

The Xilinx and Digilent websites provide myriad documents and training information for novice and expert embedded systems designers alike. As a result of this independent study I have registered for a few courses in the Xilinx University training sequence related to more advanced FPGA design methodologies.

Xilinx also provides the C and VHDL/Verilog source code for many of their IP cores. I suggest studying them because Xilinx is the "best of the best" when it comes to embedded FPGA design. An enormous amount can be learned by studying their design methodology. I also suggest the "Reuse Design Methodology Manual for SoC Design" [12]. It contains some of the best information on IP core design that I have ever found - it is an invaluable reference guide and no embedded systems designer should leave home without it.

Linux is very popular in embedded systems development since it is open-source, stable, and capable of networking. At the time of this writing, the Xilinx V2P and V4FX FPGAs support eleven completely different RTOS vendors [37]. Of course, these devices also support any completely customized operating system that is within the constraints of the FPGA and development board hardware. Should any future student wish to learn or develop their own embedded system with this board, or any other, there are a limitless number of options from which to choose.

A recent survey conducted by EE Times [76] showed that there are still mixed feelings towards the use of embedded Linux in industry. Some companies prefer the flexibility and low cost of open source software while others dismiss it due to driver incompatibility issues and lack of technical support.

I believe that open source design will become increasingly prominent in embedded systems design. As the complexity of hardware and software increases so will the process of maintaining and developing those systems. In those situations, without any key to unlock the original source code, some designs could turn into impossible tasks. The open source paradigm has allowed embedded system designers to efficiently implement and debug non-trivial applications within dynamic environments.

11 References

[1] Saini, M.. Unleash Your Creativity with Embedded Linux on Virtex-II Pro FPGAs. http://www.xilinx.com/publications/xcellonline/xcell_48/xc_pdf/xc_linux48.pdf. Winter 2004.

[2] Donaldson, J. W.. *RIT CS Graduate Independent Study*. https://rm-rfroot.net/xupv2p/. August 2006.

[3] Carithers. W. R. *Homepage*. http://www.cs.rit.edu/~wrc/. August 2006.

[4] Lin. J.. *Jamie's Research @ WSU*. http://www.eecs.wsu.edu/~jamie/. May 2006.

[5] Van Essen. B. C. *UW CSE Homepage*. http://www.cs.washington.edu/homes/vanessen/. April 2005.

[6] Xilinx Inc.. *The Programmable Logic Company*. http://www.xilinx.com. August 2006.

[7] MontaVista Software, Inc. http://www.mvista.com/. August 2006.

[8] Xilinx Inc.. Virtex-II Pro Development System. http://www.xilinx.com/univ/xupv2p.html. August 2006.

[9] Xilinx Inc.. *Platform Studio and the EDK*. http://www.xilinx.com/ise/embedded_design_prod/platform_studio.htm. August 2006.

[10] Xilinx Inc.. *ISE Foundation*. http://www.xilinx.com/ise/logic_design_prod/foundation.htm. August 2006.

[11] Force10 Networks. *MetaTraffic Processor*. http://www.force10networks.com/products/p-series_overview.asp. August 2006.

[12] M. Keating et al. *Reuse Methodology Manual for System-On-A-Chip Designs*. June 2005.

[13] Xilinx Inc. *Microblaze Processor Central: MicroBlaze Soft Processor Core* http://www.xilinx.com/xlnx/xebiz/designResources/ip_product_details.jsp?key=micro_blaze

[14] IBM. *Cell Broadband Engine Resource Center*. http://www-128.ibm.com/developerworks/power/cell/. August 2006.

[15] MetaNetworks Technologies, Inc. http://www.metanetworks.org/. September 2005

[16] Snort. http://www.snort.org/. April 2006.

[17] Lockheed Martin Corp. *Sandia National Laboratories*. http://www.sandia.gov/. August 2006.

[18] Donaldson. J. W. *IPv6 IDS/IPS Solutions for High-Speed Networks*. https://rm-rfroot.net/masters/. August 2006.

[19] Google Inc.. http://www.google.com/. August 2006.

[20] Rochester Institute of Technology. College of Applied Science and Technology. http://www.rit.edu/~700www/. August 2006.

[21] Rochester Institute of Technology. http://www.rit.edu/. August 2006.

[22] Rochester Institute of Technology. *Department of Electrical, Computer, & Telecommunications Engineering Technology*. http://www.rit.edu/%7E706www/. August 2006.

[23] Digilent Inc.. *Xilinx Development Boards Reseller*. http://digilentinc.com/. April 2006.

[24] Digilent Inc.. XUPV2P Built-In Self Test. http://www.digilentinc.com/Data/Products/XUPV2P/bist_rev_1_5.zip. August 2006.

[25] Xilinx Inc.. *Virtex-II Pro Development System Hardware Reference Manual*. http://direct.xilinx.com/bvdocs/userguides/ug069.pdf. March 2005.

[26] Digilent Inc. *XUP-V2Pro Pack*. http://www.digilentinc.com/Data/Products/XUPV2P/EDK-XUP-V2ProPack.zip

[27] Xilinx Inc.. *Embedded Design: Embedded Operating Systems Support*. http://www.xilinx.com/products/design_resources/design_tool/grouping/embedded_os.htm July 2006.

[28] Xilinx Inc. *Virtex-II Pro Family Product Table*. http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex_ii_pro_fpgas/product_table.htm

[29] Xilinx Inc.. *Black Box Component Generation*. http://www.xilinx.com/xlnx/xil_ans_display.jsp?BV_UseBVCookie=yes&getPagePath=9838&iLanguageID=1. October 2005.

[30] Xilinx Inc.. *CORE Generator*. http://www.xilinx.com/ise/products/coregen_overview.pdf. August 2006.

[31] Kegel, D.. *Crosstool*. http://kegel.com/crosstool/#download. March 2006.

[32] Xilinx Inc.. *Embedded System Tools Reference Manual: Embedded Development Kit* 8.1i. http://www.xilinx.com/ise/embedded/est_rm.pdf. October 2005.

[33] GNU Software. http://www.gnu.org/software/screen/. December 2005.

[34] GNU Software. http://samba.anu.edu.au/rsync/. April 2006.

[35] BitMover Inc.. http://www.bitkeeper.com/. August 2006.

[36] Xilinx Inc.. *Generating ACE File in EDK*. ftp://ftp.xilinx.com/pub/documentation/misc/system_ace.pdf.

[37] BusyBox.net. *The Swiss Army Knife of Embedded Linux*. http://www.busybox.net/downloads/BusyBox.html. August 2006.

[38] Klingauf, W.. *Linux on the Xilinx ML300*. http://www.klingauf.de/v2p/index.phtml. January 2004.

[39] BusyBox.net. *udhcp Server/Client Package*. http://udhcp.busybox.net/. October 2002.

[40] Embedded.com. *Introduction to the Volatile Keyword*. http://www.embedded.com/story/OEG20010615S0107. July 2001.

[41] LinuxReviews.com. *Jobs – The Basics of Job Control*. http://linuxreviews.org/beginner/jobs/. 2005.

[42] PenguinPPC: *Embedded Systems*. http://penguinppc.org/embedded/

[43] Moolenaar, B.. *Vim User Manual*. http://vim.dindinx.net/orig/html/usr_23.txt.php. September 2003.

[44] Tech-FAQ.com. *What is ECC memory?* http://www.tech-faq.com/ecc-memory.shtml

[45] RealWorldTech.com. *Parity and ECC – How They Work*. http://www.realworldtech.com/page.cfm?ArticleID=RWT061300000000. June 2000.

[46] Kingston Technology Co. *Memory Ranks and Intel E7320/E7520 Chipset Based Servers*. http://www.kingston.com/ukroot/serverzone/pdf_files/Mem_ranks_ENG.pdf. April 2005.

[47] Transtronics Inc.. *How Memory Works*. http://www.xtronics.com/memory/how_memory-works.htm. August 2006.

[48] PCvsConsole.com. *Quad Pumped FSB and DDR Memory Bus*. http://www.pcvsconsole.com/hank/answer.php?file=571. March 2004.

[49] Xilinx Inc.. *Xilinx XUP Virtex-II Pro Development System: Supported DIMM Modules*. http://www.xilinx.com/univ/xupv2p_supported_dimms.html. August 2006.

[50] PCper.com. *Memory: From Basics to Buying*. http://www.pcper.com/article.php?aid=-279&type=expert. February 2004.

[51] PCI-SIG. *PCI Express®: Performance Scalability for the Next Decade*. http://www.pcisig.com/specifications/pciexpress/. August 2006.

[52] Infiniband Trade Association. http://www.infinibandta.org/home. Ausut 2006.

[53] Wikipedia. *Self-clocking Signal*. http://en.wikipedia.org/wiki/Self-clocking_signal. June 2006.

[54] Micron Inc.. *General DDR SDRAM Functionality*. http://download.micron.com/pdf/technotes/TN4605.pdf. July 2001.

[55] TLDP.org. *The Unix and Internet Fundamentals HOWTO*. http://en.tldp.org/HOWTO/Unix-and-Internet-Fundamentals-HOWTO/. March 2004.

[56] TLDP.org. *From Power Up to Bash Prompt*. http://www.tldp.org/HOWTO/From-PowerUp-To-Bash-Prompt-HOWTO.html. November 2000.

[57] Wikipedia.org. *BSD and GPL Licensing*. http://en.wikipedia.org/wiki/BSD_and_GPL_licensing. August 2006.

[58] IBM.com. *IBM CoreConnect Bus Cores*. http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/F175B826ECE6FDE08725711F00770F60/$file/G224-7587-01_coreconnect_pb.pdf. August 2006.

[59] IBM.com. *CoreConnect Bus Architecture*. http://www-03.ibm.com/chips/products/coreconnect/. August 2006.

[60] Xilinx Inc.. *CoreConnect Technology Xilinx IP Core*. http://www.xilinx.com/xlnx/xebiz/designResources/ip_product_details.jsp?key=dr_pcentral_coreconnect. August 2006.

[61] Globe2000. *Tutorial Assembler*. http://www.giobe2000.it/Tutorial/img/Cap1_1/8086.gif. August 2006.

[62] Xilinx Inc.. *Using Delay-Locked Loops in Spartan-II FPGAs*. http://www.xilinx.com/bvdocs/appnotes/xapp174.pdf. January 2000.

[63] Altera Corp.. *Jitter Comparison Analysis*. http://www.altera.com/literature/tb/tb70.pdf. January 2001.

[64] Saleh. R.. *Flip-Flop and Clock Design*. http://www.ece.ubc.ca/~elec579/clockflop.pdf. August 2006.

[65] Xilinx Inc.. *Xilinx University Program: Lab 1: Software Flow*. http://www.cs.unb.ca/courses/cs4805/assignments/ISEflow.pdf. August 2006.

[66] IEEE. 802.3 CSMA/CD (Ethernet). http://grouper.ieee.org/groups/802/3/. August 2006.

[67] Wikipedia.org. *Media Access Control*. http://en.wikipedia.org/wiki/Media_Access_Control. August 2006.

[68] Wikipedia.org. *Media Independent Interface*. http://en.wikipedia.org/wiki/Media_Independent_Interface. August 2006.

[69] Intel Corp. *LXT972A Single-Port 10/100 Mbps PHY Transceiver*. http://www.intel.com/design/network/products/lan/datashts/24918603.pdf. August 2006.

[70] *Understanding the Root File System* http://www.unet.univie.ac.at/aix/aixbman/admnconc/root_fs_under.htm

[71] Hauser. J.. *SoftFloat*. http://www.jhauser.us/arithmetic/SoftFloat.html. August 2006.

[72] MIT. *MTL Laboratories*. http://www-mtl.mit.edu/Courses/6.111/labkit/ise.shtml. August 2006.

[73] Sullins, D.. *Dave's Software*. http://home.earthlink.net/~davesullins/software/bitinfo-0.3.tar.gz. August 2006.

[74] Opencores.org. http://www.opencores.org/. August 2006.

[75] Johnston, M.. *Dropbear SSH Server and Client*. http://matt.ucc.asn.au/dropbear/. August 2006.

[76] EE Times. *2006 Embedded Systems Design State of Embedded Survey*. http://www.eet.com/news/semi/showArticle.jhtml?articleID=184425894. April 2006.

[77] Evans, K. S.. Transition Planning for Internet Protocol Version 6 (IPv6). http://www.whitehouse.gov/omb/memoranda/fy2005/m05-22.pdf. August 2005.

[78] Red Hat Inc.. The Open Source Leader. http://www.redhat.com/. August 2006.

[79] Sun Microsystems. Solaris 9 Operating Environment. http://www.sun.com/software/solaris. August 2006.

[80] The Eclipse Foundation. http://www.eclipse.org/. August 2006.

Appendix

A.1 Minimum Kernel Configuration w/o RFS

Below are the minimum (i.e. anything not listed here may be disabled) kernel configuration options that must be enabled in order to boot the kernel image and ensure the hardware is at least partially functional. This file may be downloaded from my website [2] – it will be named '*.config.MIN.noRFS*'.

```
Platform Support --->
      (40x) Processor Type
      (Xilinx-ML300) Machine Type
      [*] Math emulation
      (UART0) TTYS0 device and default console
General Setup --->
      [*] Default bootloader kernel arguments
            …string: "console=ttyS0,9600"
Character devices --->
      [*] Standard/generic (8250/16550 and compatible UARTs) serial support
      [*]   Support for console on serial port
```

Be sure that the serial console's baud rate is set properly in the kernel command string (shown in bold) otherwise there will be no output (or gibberish) displayed to the console window.

Note that we require math emulation as the PPC405GP lacks a math coprocessor for floating point operations. Kernel math emulation is quite slow compared to a software FP-enabled userland. Hence, try something like SoftFloat [71] if higher performance is required. Of course, there is always the option of buying a FPU ASSP/CSSP to attach to the board. But the interface controller would have to be written manually in either VHDL or Verilog.

I believe the other options are self-explanatory, but for more information on any of these options refer to the "HELP" dialog in the kernel menu or my personal favorite, **Google**. As one can see, very few kernel configuration options need be enabled for proper booting.

A.2 Kernel Boot Log w/o RFS

Below is the log from my initial boot test without having any RFS installed in the CF card. We can see that the kernel is first copied into main memory (compare addresses shown to those in the memory map) and then begins the main boot cycle.

```
loaded at:     00400000 0044C1E4
board data at: 0044913C 00449154
relocated to:  00405618 00405630
zimage at:     00405B1D 0044831A
avail ram:     0044D000 10000000

Linux/PPC load: console=ttyS0,9600
Uncompressing Linux...done.
```

```
Now booting the kernel
Linux version 2.4.26 (jwdonal@terranova) (gcc version 3.4.4) #52 Thu
Aug 3 23:15:34 MDT 2006
Xilinx Virtex-II Pro port (C) 2002 MontaVista Software, Inc.
(source@mvista.com)
On node 0 totalpages: 65536
zone(0): 65536 pages.
zone(1): 0 pages.
zone(2): 0 pages.
Kernel command line: console=ttyS0,9600
Xilinx INTC #0 at 0x41200000 mapped to 0xFDFFE000
Calibrating delay loop... 299.82 BogoMIPS
Memory: 258352k available (500k kernel code, 176k data, 28k init, 0k
highmem)
Dentry cache hash table entries: 32768 (order: 6, 262144 bytes)
Inode cache hash table entries: 16384 (order: 5, 131072 bytes)
Mount cache hash table entries: 512 (order: 0, 4096 bytes)
Buffer cache hash table entries: 16384 (order: 4, 65536 bytes)
Page-cache hash table entries: 65536 (order: 6, 262144 bytes)
POSIX conformance testing by UNIFIX
Linux NET4.0 for Linux 2.4
Based upon Swansea University Computer Society NET3.039
Starting kswapd
Serial driver version 5.05c (2001-07-08) with no serial options
enabled
ttyS00 at 0xfdfff003 (irq = 29) is a 16550A
Kernel panic: VFS: Unable to mount root fs on 00:00
 <0>Rebooting in 180 seconds..
```

Clearly, the kernel is stating that there is no root file system available for mounting. Thus, there is no '`init`' process, no way to attach to a virtual terminal, and no way to login to a shell. This error will be remedied with BusyBox.

A.3 Minimum BusyBox Configuration

Following is the minimum (i.e. everything else may be disabled) BusyBox configuration to receive a shell prompt after the kernel boots – rather than a nasty kernel panic message. One will certainly want to add many more features than what is shown below to make the OS more developer-friendly. My personal configuration (not the minimum) can be found on my website [2].

```
Busybox Settings  --->
      Build Options --->
            [*] Do you want to build BusyBox with a Cross…
            (powerpc-405-linux-gnu-) Cross Compiler prefix
      Installation Options --->
      (<user_home_dir>/<RFS_dir>) BusyBox installation prefix
Init Utilities --->
      [*] init
      [*]   Support reading an inittab file
```

```
    Login/Password Management Utilities --->
        [*] getty
        [*] login
    Shells --->
        Choose your default shell (ash)  --->
        --- ash
```

If using the '`/etc`' directory off of my website [2] some of the files assume that extra options have been enabled. Be sure to check every file in the '/etc' directory as I have listed any assumptions in each file's comment header. I also strongly encourage the inclusion of the '`cat`' command as it is useful for retrieving information from the kernel post-boot through the '`/proc`' virtual file system. Remember, customization of at least some files in '`/etc`' will be desired.

When choosing the default shell one should fully understand that the BusyBox binary is compiled for a "multi-call" environment. For example, whenever '`/bin/sh`' is called, BusyBox is executed and links to the default shell that was setup during the BusyBox configuration with '`make menuconfig`'. A similar effect occurs with all of the BusyBox processes.

As a more specific example, the default shell specified in the BusyBox configuration will not show up in the '`passwd`' file when using '`adduser`' (i.e. it will still show '`/bin/sh`' as the default shell). However, logging into BusyBox will result in the following message:

```
BusyBox v1.2.0 (2006.08.02-06:13+0000) Built-in shell (ash)
```

Notice that it has still started the 'ash' shell regardless of what the '*`passwd`*' file states. If, however, the '*`passwd`*' file is changed to explicitly run another shell (e.g. hush) then that shell will be run no matter what the default shell is set to – provided that the shell was installed.

A.4 Minimum Kernel Configuration w/ RFS

Below is the minimum kernel configuration required to mount the BusyBox RFS. This does not, however, include enough support required for the '`/etc`' directory files on my website. For those configurations please refer to the *Tutorial* or download the configuration file from my website. The file on my website will be called '`.config_RFS_SHM`' as it adds additional support for "SHared Memory".

```
Code maturity level options  --->
    [*] Prompt for development and/or incomplete code/drivers
Platform Support --->
    (40x) Processor Type
    (Xilinx-ML300) Machine Type
    [*] Math emulation
    (UART0) TTYS0 device and default console
General Setup --->
    [*] Default bootloader kernel arguments
```

```
          …string: "console=ttyS0,9600
          root=/dev/xsysace/disc0/part3 rw"
Block devices --->
     [*] Xilinx on-chip System ACE
Character devices --->
     [*] Standard/generic (8250/16550 and compatible UARTs)
serial support
     [*]   Support for console on serial port
File systems --->
     [*] /proc file system support
     [*] /dev file system support (EXPERIMENTAL)
     [*]   Automatically mount at boot
     [*] Second extended fs support
```

A.5 Kernel Boot Log w/ RFS

A boot log of the kernel mounting the RFS can be found below. I have highlighted (in bold) the most important changes that have occurred in this boot log since that which was shown in *Appendix* section A.2

```
loaded at:     00400000 004591E4
board data at: 0045613C 00456154
relocated to:  00405638 00405650
zimage at:     00405B3D 004552AE
avail ram:     0045A000 10000000

Linux/PPC load: console=ttyS0,9600 root=/dev/xsysace/disc0/part3 rw
Uncompressing Linux...done.
Now booting the kernel
Linux version 2.4.26 (jwdonal@terranova) (gcc version 3.4.4) #63 Fri
Aug 4 01:47:34 M
DT 2006
Xilinx Virtex-II Pro port (C) 2002 MontaVista Software, Inc.
(source@mvista.com)
On node 0 totalpages: 65536
zone(0): 65536 pages.
zone(1): 0 pages.
zone(2): 0 pages.
Kernel command line: console=ttyS0,9600 root=/dev/xsysace/disc0/part3
rw
Xilinx INTC #0 at 0x41200000 mapped to 0xFDFFE000
Calibrating delay loop... 299.82 BogoMIPS
Memory: 258244k available (588k kernel code, 192k data, 32k init, 0k
highmem)
Dentry cache hash table entries: 32768 (order: 6, 262144 bytes)
Inode cache hash table entries: 16384 (order: 5, 131072 bytes)
Mount cache hash table entries: 512 (order: 0, 4096 bytes)
Buffer cache hash table entries: 16384 (order: 4, 65536 bytes)
Page-cache hash table entries: 65536 (order: 6, 262144 bytes)
```

```
POSIX conformance testing by UNIFIX
Linux NET4.0 for Linux 2.4
Based upon Swansea University Computer Society NET3.039
Starting kswapd
devfs: v1.12c (20020818) Richard Gooch (rgooch@atnf.csiro.au)
devfs: boot_options: 0x1
Serial driver version 5.05c (2001-07-08) with no serial options
enabled
ttyS00 at 0xfdfff003 (irq = 29) is a 16550A
Partition check:
 xsysacea: p1 p2 p3
System ACE at 0x41800000 mapped to 0xD1000000, irq=30, 1000944KB
VFS: Mounted root (ext2 filesystem).
Mounted devfs on /dev
Freeing unused kernel memory: 32k init
init started:  BusyBox v1.2.0 (2006.08.04-07:57+0000) multi-call
binary

MontaVista PPC Linux 2.4.26
xupv2p login:
```

For more information on what is happening in the boot log please refer to the *Lessons Learned* section on the Linux boot process.

A.6 File Descriptions

The numerous file extensions within any ISE or XPS project can be quite intimidating. I have included a *short* list of some Xilinx file extensions here to make the task of understanding the design and synthesis flow a bit less daunting. For the "decode" of each file extension acronym please reference the Acronyms section of the *Appendix*. The majority of this information was taken from the Xilinx help files and MTL laboratories at MIT [72].

EDIF – the industry standard file format for specifying a design netlist.

UCF – contains the design constraints (signal-to-pin assignments, timing parameters, etc) specified by the user prior to synthesis.

NCF – holds the constraints created by the schematic editor and/or 3rd party tools.

SRP – describes the logic inferred for each part of the HDL file, and often includes helpful warning messages.

NGC – replaces both the EDIF and NCF files once the synthesis tool has been used.

NGD – describes the design but is reduced to Xilinx-only primitives. NOTE: Xilinx primitives do **not** only include simple things like AND, OR, and D-registers, but could also include more complicated things like barrel shifters and arithmetic units.

NCD – contains all necessary information to implement the design on the specified device (V2P, CR-II, etc).

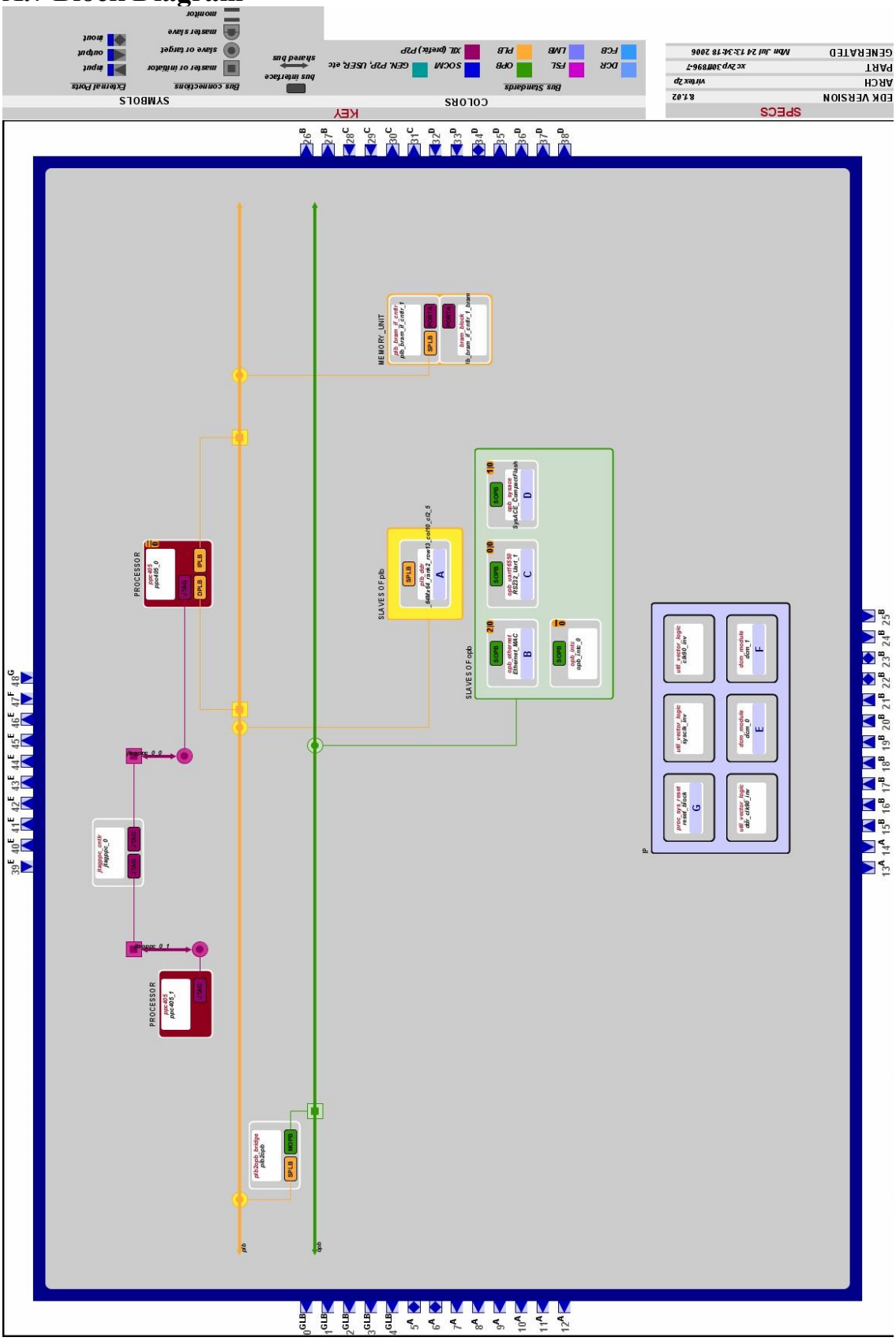
BIT – this file is simply a compilation of the following: target device, NCD file, creation date and time, bit stream length, and (if using XPS) the ELF binary of the user's application that is to be loaded into BRAM. The interested reader can download a BIT file parser at [73].

ELF – this is the final executable format for any user applications to be run on the PowerPC processor.

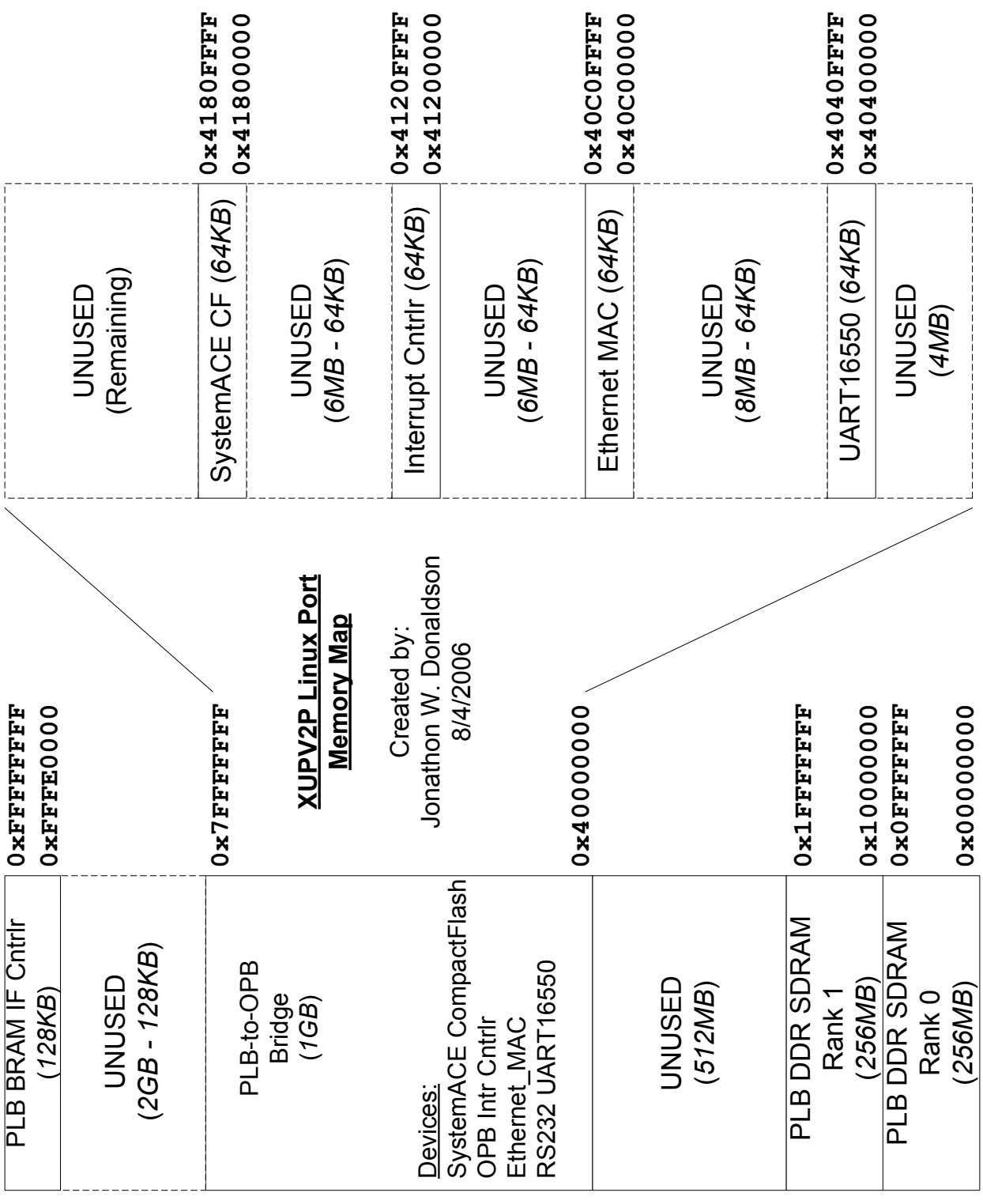
SVF – a simple text file containing both programming instructions (ELF) and configuration (BIT) data for JTAG operations.

ACE – format created from the final SVF file for proper parsing by the Xilinx SystemACE controller.

A.7 Block Diagram



A.8 Memory Map



A.9 Acronyms

Acronym	Decode
ACE	Adaptive Configuration Environment
AGP	Accelerated Graphics Port
AHB	Advanced High-performance Bus
ALU	Arithmetic Logic Unit
AMBA	Advanced Microcontroller Bus Architecture
API	Application Programming Interface
APU	Arithmetic Processing Unit
ARM	Advanced RISC Machine
ASIC	Application Specific Integrated Circuit
ASSP	Application Specific Standard Product
AUI	Attachment Unit Interface
BASH	Bourne Again SHell
BIOS	Basic Input/Output System
BRAM	Block RAM
BSB	Base System Builder
BSP	Board Support Package
BSS	Block Started by Symbols
BST	Board Support Toolkit
CAST	College of Applied Science and Technology
CB	Check Bit
CBE	Cell Broadband Engine
CCB	CoreConnect™ Bus
CET	Computer Engineering Technology
CF	Compact Flash
CFI	Common Flash Interface
CMOS	Complimentary Metal-Oxide Semiconductor
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
CR-II	CoolRunner-II
CSMA/CD	Carrier Sense Multiple Access/Collision Detect
CSSP	Customer Specific Standard Product
DCM	Digital Clock Manager
DCR	Device Control Register bus
DDR	Double Data Rate
DHCP	Dynamic Host Control Protocol
DIMM	Dual Inline Memory Modules
DLL	Delay-Locked Loop
DMA	Direct Memory Access

DMAC	DMA Controller
DNU	Do Not Use
DOS	Disk Operating System
DRAM	Dynamic Random Access Memory
ECC	Error-Correction Code
EDIF	Electronic Data Interchange Format
EDK	Embedded Development Kit
EEPROM	Electrically Erasable PROM
ELDK	Embedded Linux Development Kit
ELF	Executable and Linking Format
EMAC	Ethernet MAC
EOL	End of Line
eOS	embedded Operating System
ESD	Embedded Systems Design
FCB	FIFO Control Block
FF	Flip-Chip Fine-Pitch
FIFO	First In First Out
FIT	Fixed Interval Timer
FLP	Fast Link Pulse
FNF	File Not Found
FP	Floating Point
FPGA	Field Programmable Gate Array
FPM	Fast Page Mode
FPU	Floating Point Unit
FSB	Front Side Bus
FSL	Fast Simplex Link
GCC	GNU Compiler Collection
GDB	GNU Debugger
GMII	Gigabit MII
GNU	Gnu's Not UNIX
GPIO	General Purpose Input/Output
GPL	General Public License
GPL	General Public License
GTL	Gunning Transceiver Logic Terminated
GTLP	GTL Plus
GUI	Graphical User Interface
HSTL	High Speed Transceiver Logic
HTTP	Hyper Text Transfer Protocol
I2C	IIC
IBM	International Business Machines
IC	Integrated Circuit

IDE	Integrated Development Environment
IIC	Inter-Integrated Circuit
IP	Intellectual Property
IP	Internet Protocol
IPC	Intel Process Communications
IPIF	Intellectual Property Interface
IPv4	IP version 4
IPv6	IP version 6
IRQ	Interrupt Request Line
ISA	Instruction Set Architecture
ISE	Integrated Software Environment™
ISP	Internet Service Provider
JEDEC	Joint Electron Device Engineering Council
JTAG	Joint Test Action Group
LGPL	Lesser GPL
LLC	Logical Link Control
LMA	Load Memory Address
LMB	Local Memory Bus
LSP	Linux Support Package
LVCMOS	Low Voltage CMOS
LVDS	Low Voltage Differential Signaling
LVTTL	Low Voltage TTL
MAC	Media Access Controller
MCH	Memory Controller Hub
MDD	Microprocessor Driver Description
MGT	Multi-Gigabit Transceiver
MHS	MicroProcessor Hardware Spec
MII	Media Independent Interface
MIPS	Millions of Instructions Per Second
MLD	Microprocessor Library Definition
MMU	Memory Management Unit
MPD	MicroProcessor Peripheral Description
MSR	Machine State Register
MSS	MicroProcessor Software Spec
NCD	Native Circuit Description
NGC	Netlist Gates and Constraints
NGD	Native Generic Database
NLA	Next Level Aggregator
OCM	On-Chip Memory
OMB	Office of Management and Budget
OPB	On-Chip Peripheral Bus

OS	Operating System
PACE	Pinout and Area Constraints Editor
PAO	Peripheral Analysis Order
PAR	Place and Route
PC4	PCIV
PCB	Printed Circuit Board
PCI	Peripheral Component Interconnect
PCI-E	PCI-Express
PCIV	Parallel Cable IV
PHY	Physical Layer Transceiver
PID	Process IDentification
PIT	Programmable Interval Timer
PLB	Processor Local Bus
PLL	Phase-Locked Loop
POSIX	Potable Operating System Interface
PPC	PowerPC
PPC405GP	PPC 405 General Purpose
PPT	PowerPoinT
PROM	Programmable Read-Only Memory
PSM	POSIX Shared Memory
QDR	Quadruple Data Rate
RAM	Random Access Memory
RDIMM	Registered DIMM
RFC	Request For Comments
RFS	Root File System
RISC	Reduced Instruction Set Computer
RIT	Rochester Institute of Technology
RTC	Real-Time Clock
RTOS	Real-Time Operating System
SATA	Serial Advanced Technology Attachment
SDR	Single Data Rate
SDRAM	Synchronous Dynamic RAM
SEEPROM	Serial EEPROM
SER	Soft Error Rate
SHM	SHared Memory
SMA	Sub-Miniature A
SOC	System On a Chip
SOCM	SOC Model
SPI	System Packet Interface
SPLD	Simple Programmable Logic Device
SRP	Synthesis Report

SSH	Secure SHell
SSTL	Stub Series Terminated Logic
SVF	Serial Vector Format
TCL	Tool Command Language (µPC Driver Command)
TLA	Top Level Aggregator
TTL	Transistor-Transistor Logic
UART	Universal Asynchronous Receiver/Transmitter
UCF	User Constraints File
UDIMM	Unregistered DIMM
USB	Universal Serial Bus
V2P	Virtex-II Pro
V4FX	Virtex-4 FX
VCC	Voltage Collector-Collector
VDD	Voltage Drain-Drain
VGA	Video Graphics Array
VHDL	VHSIC Hardware Description Language
VhMC	VHDL Model Compiler
VHSIC	Very High Speed Integrated Circuit
VMA	Virtual Memory Address
VMC	Verilog Model Compiler
VSS	Voltage Substrate and Sources (Ground)
WDT	Watchdog Timer
WFS	Web File Share
XAUI	10Gb AUI
XDR	Xtreme Data Rate
XMD	Xilinx MicroProcessor Debugger
XMK	Xilinx Micro Kernel
XPS	Xilinx Platform Studio
XST	Xilinx Synthesis Technology
XUPV2P	Xilinx University Program V2P
ZPR	Zone Protection Register